Julia Language Documentation

Versão 0.2.0

Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman, et al.

23/08/2014

Sumário

CAPÍTULO 1

O Manual de Julia

Versão 0.2

Data 23/08/2014

Versão do original 5 de Abril, 2013

1.1 Introdução

Computação científica tem requerido, tradicionalmente, alta performace embora grandes nomes da área tenham passado a utilizar linguagens dinâmicas lentas para o trabalho diário. Acreditamos que existam várias boas razões para preferir utilizar linguagens dinâmicas em suas aplicações, e não esperamos desmerecer seu uso. Felizmente, as modernas técnicas para criação de linguagens e de compilação torna possível eliminar, quase totalmente, o problema de desempenho de linguagens dinâmicas e prover um ambiente produtivo para experimentação e eficiente para produção de aplicativos que precisam de alto desempenho. A linguagem de programação Julia preenche esse buraco: é uma linguagem dinâmica, apropriada para computação numérica e científica, com um desempenho comparável a linguagens estáticas tradicionalmente utilizadas.

As características de Julia são tipagem opcional, *multiple dispatch*, e bom desempenho, alcançado utilizando inferência de tipos e compilação *just-in-time* (JIT) [1], [2], implementada utilizando *LLVM* [3], [4]. Ela é multi-paradigma, combinando características de programação imperativa, funcional e orientada a objetos. A sintaxe de Julia é similar a do GNU Octave ou MATLAB(R) e consequentemente os programadores que que já utilizam estas linguagens devem sentir-se imediatamente confortáveis com Julia. Enquanto MATLAB(R) é um bem eficiente para experimentações e explorações de álgebra linear numérica, possui limitações para tarefas computacionais fora deste campo relativamente pequeno. Julia mantem a facilidade e expressividade do MATLAB(R) para computação numérica de alto nível, mas ultrapassa as limitações comparadas a uma linguagem de programação de propósito geral. Para alcançar isso, Julia é construída com heranças das linguagens de programação matemática, mas também herda muito de outras linguagens dinâmicas populares, incluindo Lisp, Perl, Python, Lua, and Ruby.

As características mas significativas de Julia em relação a linguagens dinâmicas típicas são:

- O núcleo da linguagem impõe muito pouco; a biblioteca padrão é escrita utilizando a própria linguagem Julia, incluindo operadores primitivos como operações aritméticas de inteiros
- Uma grande variedades de tipos para construir e descrever objetos, que pode também, opcionalmente, ser utilizado para fazer declarações de tipos

[1] http://en.wikipedia.org/wiki/Just-in-time_compilation
[2] http://pt.wikipedia.org/wiki/JIT
[3] http://en.wikipedia.org/wiki/Low_Level_Virtual_Machine
[4] http://pt.wikipedia.org/wiki/Low_Level_Virtual_Machine

- A habilidade de definir o comportamento de funções com base na combinação de vários tipos de argumentos via *multiple dispatch* [5], [6]
- Geração automática de código eficiente e especializado para diferentes tipos de argumentos
- Bom desempenho, aproximando-se de linguagens estáticas e compiladas como C

Embora alguns por vezes digam que linguagens dinâmicas não são tipadas, elas definitivamente são: todo objeto, seja primitivo ou definido pelo usuário, possui um tipo. A ausência na declaração do tipo na maioria das linguagens dinâmicas, entretanto, significa que não podemos instruir o compilador sobre o tipo dos valores, e comumente não podemos falar sobre tipos. Em linguagens estáticas, em oposição, enquanto podemos - e usualmente precisamos - especificar tipos para o compilador, tipos existem apenas em tempo de compilação e não podem ser manipulados ou expressos em tempo de execução. Em Julia, tipos são objetos em tempo de execução, e podem também ser utilizados para convenientemente informar o compilador.

Embora o programador casual não precise explicitamente utilizar tipos ou *multiple dispatch*, estas são características principais de Julia: funções são definidas para diferentes combinações de tipos de argumentos, e utilizadas de acordo com as especificações mais semelhantes. Este modelo ser para programação matemáticas, onde não é natural o primeiro argumento "possuir" uma operação como é tradicional em linguagens orientadas a objetos. Operadores são apenas funções com uma função especial - para estender a adição para um novo tipo definido pelo usuário, você define um novo método para a função +. Codes já existentes são aplicados para novos tipos sem problemas.

Parcialmente por causa da inferência de tipos em tempo de execução (aumentado pela opcionalidade da declaração de tipo), e parcialmente por causa do grande foco em desempenho existente no início do projeto, a eficiência computacional de Julia é maior que a de outras linguagens dinâmicas, e até rivaliza com linguagens estáticas e compiladas. Para problemas numéricos de larga escala, velocidade sempre foi, continua sendo, e provavelmente sempre será crucial: a quantidade de dados sendo processada tem seguido a Lei de Moore na década passada.

Julia anseia criar uma combinação sem precedente de facilidade de uso, força e eficiência em uma única linguagem. Em adição ao dito acima, algumas das vantagens de Julia em comparação com outros sistemas são:

- Livre e open source (Licença MIT)
- Tipos definidos pelo usuário são rápidos e compactos como tipos nativos
- Ausência da necessidade de vetorizar códigos por desempenho; códigos não vetorizados são rápidos
- Projetado para computação paralela e distribuída
- *Lightweight "green" threading coroutines* [7], [8]
- Sistemas de tipos não obstrutivos mas poderoso
- Conversão e promoção de tipos numéricos e outros de forma elegante e extensível
- Suporte eficiente para Unicode, incluindo mas não limitado ao UTF-8
- Chamadas de funções em C de forma direta (sem necessidade de *wrappers* ou *API* especial)
- Capacidade semelhante a de uma poderosa *shell* para gerenciar outros processos
- Macros de forma parecida a Lisp e outras facilidades de metaprogramação

1.2 Começando

A instalação de Julia é direta, seja com utilizando binário pré-compilados, seja compilando o código-fonte. Baixe e instale Julia seguindo as instruções (em inglês) em http://julialang.org/downloads/.

[5] http://en.wikipedia.org/wiki/Multiple_dispatch
[6] http://pt.wikipedia.org/wiki/Despacho_m%C3%BAltiplo
[7] http://en.wikipedia.org/wiki/Coroutine
[8] http://pt.wikipedia.org/wiki/Corotina

A maneira mais fácil de aprender e experimentar com Julia é iniciando sessão interativa (também conhecida como *read-eval-print loop* ou *"repl"* [9]):

```
$ julia
               _
   _       _ _(_)_     |
  (_)     | (_) (_)    |  A fresh approach to technical computing.
   _ _   _| |_  __ _   |
  | | | | | | |/ _` |  |  Version 0 (pre-release)
  | | |_| | | | (_| |  |  Commit 61847c5aa7 (2011-08-20 06:11:31)*
 _/ |\__'_|_|_|\__'_|  |
|__/                   |

julia> 1 + 2
3

julia> ans
3
```

Para encerrar a sessão interative, digite `^D`- a tecla *Ctrl* em conjunto da tecla `d` - ou digite `quit()`. Quando utilizando Julia no modo interativo, `julia` mostra um *banner* e espera o usuário digitar um comando. Uma vez que o usuário digitou comando, como *1 + 2*, e pressionou *enter*, a sessão interativa calcula a expressão e mostra o resultado. Se uma expressão é inserida em uma sessão interativa com um ponto-e-vírgula no final, seu resultado será calculado, mas não mostrado. A variável `ans` armazena o resultado da última expressão calculada, tendo sido mostrada ou não.

Para calcular expressões escritas em um arquivo `file.jl`, digite `include("file.jl")`.

Para rodar código em um arquivo de maneira não-interativa, você pode passar o nome do arquivo como o primeiro argumento na chamada de Julia:

```
$ julia script.jl arg1 arg2...
```

Como mostra o exemplo, os argumentos da linha de comando subsequentes são tomados como argumentos para o programa `script.jl`, passados na constante global `ARGS`. `ARGS` é também definida quando o código do *script* é dado usando a opção da linha de comando `-e` (veja a saída de ajuda de `julia` abaixo). Por exemplo, para apenas imprimir os argumentos dados a um *script*, você pode fazer:

```
$ julia -e 'for x in ARGS; println(x); end' foo bar
foo
bar
```

Ou pode colocar esse código em um *script* e rodá-lo:

```
$ echo 'for x in ARGS; println(x); end' > script.jl
$ julia script.jl foo bar
foo
bar
```

Há várias maneiras de chamar Julia e passar opções, semelhantes àquelas disponíveis para os programas `perl` e `ruby`:

```
julia [options] [program] [args...]
 -v --version             Display version information
 -q --quiet               Quiet startup without banner
 -H --home=<dir>          Load files relative to <dir>
 -T --tab=<size>          Set REPL tab width to <size>

 -e --eval=<expr>         Evaluate <expr>
```

[9] http://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop

```
-E --print=<expr>          Evaluate and show <expr>
-P --post-boot=<expr>      Evaluate <expr> right after boot
-L --load=file             Load <file> right after boot
-J --sysimage=file         Start up with the given system image file

-p n                       Run n local processes
--machinefile file         Run processes on hosts listed in file

--no-history               Don't load or save history
-f --no-startup            Don't load ~/.juliarc.jl
-F                         Load ~/.juliarc.jl, then handle remaining inputs

-h --help                  Print this message
```

1.2.1 Tutoriais

Alguns guias passo-a-passo estão disponíveis online:

- Começando com Julia para usuários de MATLAB
- Forio Julia Tutorials (em inglês)
- Tutorial for Homer Reid's numerical analysis class (em inglês)

1.2.2 Diferenças nótáveis em relação ao MATLAB

Usuários de MATLAB podem achar a sintaxe de Julia familar, porém Julia não é de maneira alguma um clone de MATLAB: há grandes diferenças sintáticas e funcionais. Apresentadas a seguir estão algumas importantes ressalvas que podem confundir usuários de Julia acostumados com MATLAB:

- *Arrays* são indexados com colchetes, `A[i,j]`.
- A unidade imaginária `sqrt(-1)` é representada em Julia por `im`.
- Múltiplos valores são retornados e atribuídos com parênteses, `return (a, b)` e `(a, b) = f(x)`.
- Valores são passados e atribuídos por referência. Se uma função modifica um *array*, as mudanças serão visíveis para quem chamou.
- Julia tem *arrays* unidimensionais. Vetores-coluna são de tamanho `N`, não `Nx1`. Por exemplo, `rand(N)` cria um array unidimensional.
- Concatenar escalares e *arrays* com a sintaxe `[x,y,z]` concatena na primeira dimensão ("verticalmente"). Para a segunda dimensão, ("horizontalmente"), use espaços, como em `[x y z]`. Para construir matrizes em blocos (concatenando nas duas primeiras dimensões), é usada a sintaxe `[a b; c d]` para evitar confusão.
- Dois-pontos `a:b` e `a:b:c` constroem objetos `Range`. Para construir um vetor completo, use `linspace`, ou "concatene" o intervalo colocando-o em colchetes, `[a:b]`.
- Funções retornam valores usando a palavra-chave `return`, ao invés de por citações a seus nomes na definição da função (veja *A declaração "return"* para mais detalhes).
- Um arquivo pode conter um número qualquer de funções, e todas as definições vão ser visíveis de fora quando o arquivo for carregado.
- Reduções como `sum`, `prod`, e `max` são feitas sobre cada elemento de um *array* quando chamadas com um único argumento, como em `sum(A)`.

- Funções como `sort` que operam por padrão em colunas (`sort(A)` é equivalente a `sort(A,1)`) não possuem comportamento especial para *arrays* 1xN; o argumento é retornado inalterado, já que a operação feita foi `sort(A,1)`. Para ordenar uma matriz 1xN como um vetor, use `sort(A,2)`.
- Parênteses devem ser usados para chamar uma função com zero argumentos, como em‘‘tic()‘‘ and `toc()`.
- Não use ponto-e-vírgula para encerrar declarações. Os resultados de declarações não são automaticamente impressos (exceto no prompt interativo), e linhas de código não precisam terminar com ponto-e-vírgula. A função `println` pode ser usada para imprimir um valor seguido de uma nova linha.
- Se `A` e `B` são *arrays*, `A == B` não retorna um *array* de booleanos. Use `A .== B` no lugar. O mesmo vale para outros operaores booleanos, <, >, `!=`, etc.
- Os elementos de uma coleção podem ser passados como argumentos para uma função usando `...`, como em `xs=[1,2]; f(xs...)`.
- A função `svd` de Julia retorna os valores singulares como um vetor, e não como uma matriz diagonal.

1.2.3 Diferenças notáveis em relação a R

Um dos objetivos de Julia é providenciar uma linguagem eficiente para análise de dados e programação estatística. Para usuários de Julia vindos de R, estas são algumas diferenças importantes:

- Julia usa = para atribuição. Julia não provê nenhum outro operador alternativo, como <− ou <−.
- Julia constrói vetores usando colchetes. O `[1, 2, 3]` de Julia é o equivalente do `c(1, 2, 3)` de R.
- As operações matriciais de Julia são mais parecidas com a notação matemática tradicional do que as de R. Se `A` e `B` são matrizes, então `A * B` define a multiplicação de matrizes em Julia equivalente à `A %*% B` de R. Em R, essa notação faria um produto de Hadamard (elemento a elemento). Para obter a multiplicação elemento a elemento em Julia, você deve escrever `A .* B`.
- Julia transpõe matrizes usando o operador `'`. O `A'` em Julia é então equivalente ao `t(A)` de R.
- Julia não requer parênteses ao escrever condições `if` ou loops `for`: use `for i in [1, 2, 3]` no lugar de `for (i in c(1, 2, 3))` e `if i == 1` no lugar de `if (i == 1)`.
- Julia não trata os números `0` e `1` como booleanos. Você não pode escrever `if (1)` em Julia, porque condições `if‘ só aceitam booleanos. No lugar, escreva ‘‘if true`.
- Julia não provê funções `nrow` e `ncol`. Use `size(M, 1)` no lugar de `nrow(M)` e `size(M, 2)` no lugar de `ncol(M)`.
- A SVD de Julia não é reduzida por padrão, diferentemente de R. Para obter resultados semelhantes aos de R, você deverá chamar `svd(X, true)` em uma matrix `X`.
- Julia é uma linguagem muito cautelosa em distinguir escalares, vetores e matrizes. Em R, `1` e `c(1)` são iguais. Em Julia, eles não podem ser usados um no lugar do outro. Uma consequência potencialmente confusa é que `x' * y` para vetores `x` e `y` é um vetor de um elemento, e não um escalar. Para obter um escalar, use `dot(x, y)`.
- As funções `diag()` e `diagm()` de Julia não são parecidas com as de R.
- Julia não pode atribuir os resultados de chamadas de funções no lado esquerdo de uma operação: você não pode escrever `diag(M) = ones(n)`
- Julia desencoraja popular o *namespace* principal com funções. A maior parte das funcionalidades estatísticas para Julia é encontrada em pacotes como o *DataFrames* e o *Distributions*.
  - Funções de distribuições são encontradas no pacote Distributions
  - O pacote DataFrames provê *data frames*.

  - Fórmulas para GLM devem ser escapadas: use `:(y ~ x)` no lugar de `y ~ x`.

- Julia provê enuplas e tabelas de espalhamento reais, mas as listas de R. Quando precisar retornar múltiplos itens, você tipicamente deverá utilizar uma tupla: ao invés de `list(a = 1, b = 2)`, use `(1, 2)`.
- Julia encoraja a todos usuários escreverem seus próprios tipos. Os tipos de Julia são bem mais fáceis de se usar do que os objetos S3 ou S4 de R. O sistema de *multiple dispatch* de Julia significa que `table(x::TypeA)` e `table(x::TypeB)` agem como `table.TypeA(x)` e `table.TypeB(x)` em R.
- Em Julia, valores são passados e atribuídos por referência. Se uma função modifica um *array*, as mudanças serão visíveis no lugar de chamada. Esse comportamento é bem diferente do de R, e permite que novas funções operem em grandes estruturas de dados de maneira muito mais eficiente.
- Concatenação de vetores e matrizes é feita usando `hcat` e `vcat`, não `c`, `rbind` e `cbind`.
- Um objeto `Range` `a:b` em Julia não é uma forma abreviada de um vetor como em R, mas sim um tipo especializado de objeto que é utilizado para iteração sem muito gasto de memória. Para um converter um `Range` em um vetor, você precisa cercá-lo por colchetes: `[a:b]`.
- Julia tem várias funções que podem alterar seus argumentos. For exemplo, há tanto `sort(v)` quanto `sort!(v)`.
- Em R, eficiência requer vetorização. Em Julia, quase o contrário é verdadeiro: o código mais eficiente é frequentemente o desvetorizado.
- Diferentemente de R, não há avaliação preguiçosa [10] [11] em Julia. Para a maioria dos usuários, isso significa que há poucas expressões ou nomes de coluna sem aspas.
- Julia não possui tipo `NULL`.
- Não há equivalente do `assign` ou `get` de R em Julia.

1.3 Números Inteiros e de Ponto Flutuante

Valores inteiros e de ponto flutuante são as fundações da aritmética e computação. Representações embutidas de tais valores são chamadas de primitivas numérica, enquanto representações de inteiros e de números de ponto flutuante como valores imediatos no código são conhecidas como literais numéricos. Por exemplo, `1` é um literal numérico, enquanto `1.0` é um literal de ponto flutuante; suas representações binárias na memória como objetos são primitivas numéricas. Julia provê uma grande amplitude de tipos primitivos numéricos, e um conjunto completo de operadores aritméticos e bit a bit, e também funções matemáticas padrões, são definidas sobre eles. A seguir são apresentados os tipos numéricos primitvos de Julia:

- **Tipos de inteiros:**
  - `Int8` — inteiros 8-bit com sinal variando de -2^7 a 2^7 - 1.
  - `Int8` — inteiros 8-bit sem sinal variando de 0 a 2^8 - 1.
  - `Int16` — inteiros 16-bit com sinal variando de -2^15 a 2^15 - 1.
  - `Int16` — inteiros 16-bit sem sinal variando de 0 a 2^16 - 1.
  - `Int32` — inteiros 32-bit com sinal variando de -2^31 a 2^31 - 1.
  - `Int32` — inteiros 32-bit sem sinal variando de 0 a 2^32 - 1.
  - `Int64` — inteiros 64-bit com sinal variando de -2^63 a 2^63 - 1.
  - `Int64` — inteiros 64-bit sem sinal variando de 0 a 2^64 - 1.

[10] http://pt.wikipedia.org/wiki/Avalia%C3%A7%C3%A3o_preguiçosa
[11] http://en.wikipedia.org/wiki/Lazy_evaluation

  - `Int128` — inteiros 128-bit com sinal variando de -2^127 a 2^127 - 1.
  - `Int128` — inteiros 128-bit sem sinal variando de 0 a 2^128 - 1.
  - `Bool` — valendo ou `true` (verdadeiro) ou `false` (falso), que correspondem numericamente a 1 ou 0, respectivamente.
  - `Char` — um tipo numérico de 32 bits representando um caracter Unicode (veja *Strings* para mais detalhes).
- **Tipos de ponto flutuante:**
  - `Float32` — Números de ponto flutuante 32-bit seguindo o padrão IEEE 754.
  - `Float64` — Números de ponto flutuante 64-bit seguindo o padrão IEEE 754.

Adicionalmente, estruturas para *Complex and Rational Numbers* são construídas sobre esses tipos numéricos primitivos. Todos os tipos numéricos interoperam naturalmente sem conversão de tipos explícita, graças a um sistem flexível de promoção de tipos. Esse sistema, detalhado em *Conversion and Promotion*, pode ser estendido, possibilitando que tipos numéricos definidos pelos usuários possam interoperar tão naturalmente quanto os tipos embutidos.

1.3.1 Inteiros

Literais inteiros são representados da maneira padrão:

```
julia> 1
1

julia> 1234
1234
```

O tipo padrão para um literal inteiro depende do sistema, isto é, se ele usa uma arquitetura de 32 bits ou de 64 bits:

```
# sistema 32-bit:
julia> typeof(1)
Int32

# sistema 64-bit:
julia> typeof(1)
Int64
```

Use `WORD_SIZE` para descobrir se um sistema é de 32 ou 64 bits. O tipo `Int` é um *alias* para o tipo inteiro nativo do sistema:

```
# sistema 32-bit:
julia> Int
Int32

# sistema 64-bit:
julia> Int
Int64
```

Similarmente, `Uint` é um *alias* para o tipo inteiro sem sinal nativo do sistema:

```
# sistema 32-bit:
julia> Uint
Uint32

# sistema 64-bit:
julia> Uint
Uint64
```

Literais inteiros que não conseguem ser representados usando somente 32 bits, mas podem ser representados com 64 bits sempre criam inteiros de 64 bits, independentemente do tipo de sistema:

```
# sistema 32-bit ou 64-bit:
julia> typeof(3000000000)
Int64
```

Inteiros sem sinal são inseridos e imprimidos usando o prefixo `0x` e com dígitos hexadecimais (base 16) `0-9a-f` (você também pode usar `A-F` para a inserção). O tamanho do valores sem sinal é determinado pelo número de dígitos hexadecimais utilizados:

```
julia> 0x1
0x01

julia> typeof(ans)
Uint8

julia> 0x123
0x0123

julia> typeof(ans)
Uint16

julia> 0x1234567
0x01234567

julia> typeof(ans)
Uint32

julia> 0x123456789abcdef
0x0123456789abcdef

julia> typeof(ans)
Uint64
```

Esse comportamento é basedo na observação de que quando uma pessoa usa literais hexadecimais para valores inteiros, ela tipicamente os usa para representar uma sequência de bytes fixa ao invés de apenas um valor inteiro.

Literais binários e octais também são suportados:

```
julia> 0b10
0x02

julia> 0o10
0x08
```

Os valores mínimos e máximos representáveis dos tipos numéricos primitivos (por exemplo, inteiros) são dados pelas funções `typemin` (valor mínimo) e `typemax` (valor máximo):

```
julia> (typemin(Int32), typemax(Int32))
(-2147483648,2147483647)

julia> for T = {Int8,Int16,Int32,Int64,Int128,Uint8,Uint16,Uint32,Uint64,Uint128}
         println("$(lpad(T,6)): [$(typemin(T)),$(typemax(T))]")
       end

   Int8: [-128,127]
  Int16: [-32768,32767]
  Int32: [-2147483648,2147483647]
  Int64: [-9223372036854775808,9223372036854775807]
```

```
 Int128: [-170141183460469231731687303715884105728,170141183460469231731687303715884105727]
  Uint8: [0x00,0xff]
 Uint16: [0x0000,0xffff]
 Uint32: [0x00000000,0xffffffff]
 Uint64: [0x0000000000000000,0xffffffffffffffff]
Uint128: [0x00000000000000000000000000000000,0xffffffffffffffffffffffffffffffff]
```

Os valores retornados por `typemin` e `typemax` são sempre do mesmo tipo dos argumentos dados. A expressão acima usa várias características que ainda introduziremos, incluindo *loops for*, *Strings*, and *Interpolation*, mas deve ser fácil de entender para alguém com alguma experiência em programação.

1.3.2 Números de Ponto Flutuante

Números literais de ponto flutuante são representados nos seguintes formatos padrões:

```
julia> 1.0
1.0

julia> 1.
1.0

julia> 0.5
0.5

julia> .5
0.5

julia> -1.23
-1.23

julia> 1e10
1e+10

julia> 2.5e-4
0.00025
```

The above results are all `Float64` values. There is no literal format for `Float32`, but you can convert values to `Float32` easily:

```
julia> float32(-1.5)
-1.5

julia> typeof(ans)
Float32
```

There are three specified standard floating-point values that do not correspond to a point on the real number line:

- `Inf` — positive infinity — a value greater than all finite floating-point values
- `-Inf` — negative infinity — a value less than all finite floating-point values
- `NaN` — not a number — a value incomparable to all floating-point values (including itself).

For further discussion of how these non-finite floating-point values are ordered with respect to each other and other floats, see *Numeric Comparisons*. By the IEEE 754 standard, these floating-point values are the results of certain arithmetic operations:

```
julia> 1/0
Inf
```

```
julia> -5/0
-Inf

julia> 0.000001/0
Inf

julia> 0/0
NaN

julia> 500 + Inf
Inf

julia> 500 - Inf
-Inf

julia> Inf + Inf
Inf

julia> Inf - Inf
NaN

julia> Inf/Inf
NaN
```

The `typemin` and `typemax` functions also apply to floating-point types:

```
julia> (typemin(Float32),typemax(Float32))
(-Inf32,Inf32)

julia> (typemin(Float64),typemax(Float64))
(-Inf,Inf)
```

Note that `Float32` values have the suffix `32`: ``NaN32`, `Inf32`, and `-Inf32`.

Floating-point types also support the `eps` function, which gives the distance between `1.0` and the next larger representable floating-point value:

```
julia> eps(Float32)
1.192092896e-07

julia> eps(Float64)
2.22044604925031308e-16
```

These values are `2.0^-23` and `2.0^-52` as `Float32` and `Float64` values, respectively. The `eps` function can also take a floating-point value as an argument, and gives the absolute difference between that value and the next representable floating point value. That is, `eps(x)` yields a value of the same type as `x` such that `x + eps(x)` is the next representable floating-point value larger than `x`:

```
julia> eps(1.0)
2.22044604925031308e-16

julia> eps(1000.)
1.13686837721616030e-13

julia> eps(1e-27)
1.79366203433576585e-43

julia> eps(0.0)
5.0e-324
```

As you can see, the distance to the next larger representable floating-point value is smaller for smaller values and larger for larger values. In other words, the representable floating-point numbers are densest in the real number line near zero, and grow sparser exponentially as one moves farther away from zero. By definition, `eps(1.0)` is the same as `eps(Float64)` since `1.0` is a 64-bit floating-point value.

Background and References

For a brief but lucid presentation of how floating-point numbers are represented, see John D. Cook's article on the subject as well as his introduction to some of the issues arising from how this representation differs in behavior from the idealized abstraction of real numbers. For an excellent, in-depth discussion of floating-point numbers and issues of numerical accuracy encountered when computing with them, see David Goldberg's paper What Every Computer Scientist Should Know About Floating-Point Arithmetic. For even more extensive documentation of the history of, rationale for, and issues with floating-point numbers, as well as discussion of many other topics in numerical computing, see the collected writings of William Kahan, commonly known as the "Father of Floating-Point". Of particular interest may be An Interview with the Old Man of Floating-Point.

1.3.3 Arbitrary Precision Arithmetic

To allow computations with arbitrary precision integers and floating point numbers, Julia wraps the GNU Multiple Precision Arithmetic Library, GMP. The *BigInt* and *BigFloat* types are available in Julia for arbitrary precision integer and floating point numbers respectively.

Constructors exist to create these types from primitive numerical types, or from `String`. Once created, they participate in arithmetic with all other numeric types thanks to Julia's type promotion and conversion mechanism.

```
julia> BigInt(typemax(Int64)) + 1
9223372036854775808

julia> BigInt("123456789012345678901234567890") + 1
123456789012345678901234567891

julia> BigFloat("1.23456789012345678901")
1.23456789012345678901

julia> BigFloat(2.0^66) / 3
24595658764946068821.3

julia> factorial(BigInt(40))
815915283247897734345611269596115894272000000000
```

1.3.4 Numeric Literal Coefficients

To make common numeric formulas and expressions clearer, Julia allows variables to be immediately preceded by a numeric literal, implying multiplication. This makes writing polynomial expressions much cleaner:

```
julia> x = 3
3

julia> 2x^2 - 3x + 1
10

julia> 1.5x^2 - .5x + 1
13.0
```

It also makes writing exponential functions more elegant:

```
julia> 2^2x
64
```

The precedence of numeric literal coefficients is the same as that of unary operators such as negation. So `2^3x` is parsed as `2^(3x)`, and `2x^3` is parsed as `2*(x^3)`.

You can also use numeric literals as coefficients to parenthesized expressions:

```
julia> 2(x-1)^2 - 3(x-1) + 1
3
```

Additionally, parenthesized expressions can be used as coefficients to variables, implying multiplication of the expression by the variable:

```
julia> (x-1)x
6
```

Neither juxtaposition of two parenthesized expressions, nor placing a variable before a parenthesized expression, however, can be used to imply multiplication:

```
julia> (x-1)(x+1)
type error: apply: expected Function, got Int64

julia> x(x+1)
type error: apply: expected Function, got Int64
```

Both of these expressions are interpreted as function application: any expression that is not a numeric literal, when immediately followed by a parenthetical, is interpreted as a function applied to the values in parentheses (see *Funções* for more about functions). Thus, in both of these cases, an error occurs since the left-hand value is not a function.

The above syntactic enhancements significantly reduce the visual noise incurred when writing common mathematical formulae. Note that no whitespace may come between a numeric literal coefficient and the identifier or parenthesized expression which it multiplies.

Syntax Conflicts

Juxtaposed literal coefficient syntax conflicts with two numeric literal syntaxes: hexadecimal integer literals and engineering notation for floating-point literals. Here are some situations where syntactic conflicts arise:

- The hexadecimal integer literal expression `0xff` could be interpreted as the numeric literal `0` multiplied by the variable `xff`.
- The floating-point literal expression `1e10` could be interpreted as the numeric literal `1` multiplied by the variable `e10`, and similarly with the equivalent `E` form.

In both cases, we resolve the ambiguity in favor of interpretation as a numeric literals:

- Expressions starting with `0x` are always hexadecimal literals.
- Expressions starting with a numeric literal followed by `e` or `E` are always floating-point literals.

1.4 Mathematical Operations

Julia provides a complete collection of basic arithmetic and bitwise operators across all of its numeric primitive types, as well as providing portable, efficient implementations of a comprehensive collection of standard mathematical functions.

1.4.1 Arithmetic and Bitwise Operators

The following arithmetic operators are supported on all primitive numeric types:

- `+x` — unary plus is the identity operation.
- `-x` — unary minus maps values to their additive inverses.
- `x + y` — binary plus performs addition.
- `x - y` — binary minus performs subtraction.
- `x * y` — times performs multiplication.
- `x / y` — divide performs division.

The following bitwise operators are supported on all primitive integer types:

- `~x` — bitwise not.
- `x & y` — bitwise and.
- `x | y` — bitwise or.
- `x $ y` — bitwise xor.
- `x >>> y` — logical shift right.
- `x >> y` — arithmetic shift right.
- `x << y` — logical/arithmetic shift left.

Here are some simple examples using arithmetic operators:

```
julia> 1 + 2 + 3
6

julia> 1 - 2
-1

julia> 3*2/12
0.5
```

(By convention, we tend to space less tightly binding operators less tightly, but there are no syntactic constraints.)

Julia's promotion system makes arithmetic operations on mixtures of argument types "just work" naturally and automatically. See *Conversion and Promotion* for details of the promotion system.

Here are some examples with bitwise operators:

```
julia> ~123
-124

julia> 123 & 234
106

julia> 123 | 234
251

julia> 123 $ 234
145

julia> ~uint32(123)
0xffffff84
```

```
julia> ~uint8(123)
0x84
```

Every binary arithmetic and bitwise operator also has an updating version that assigns the result of the operation back into its left operand. For example, the updating form of + is the += operator. Writing `x += 3` is equivalent to writing `x = x + 3`:

```
julia> x = 1
1

julia> x += 3
4

julia> x
4
```

The updating versions of all the binary arithmetic and bitwise operators are:

```
+=  -=  *=  /=  &=  |=  $=  >>>=  >>=  <<=
```

1.4.2 Numeric Comparisons

Standard comparison operations are defined for all the primitive numeric types:

- == — equality.
- != — inequality.
- < — less than.
- <= — less than or equal to.
- > — greater than.
- >= — greater than or equal to.

Here are some simple examples:

```
julia> 1 == 1
true

julia> 1 == 2
false

julia> 1 != 2
true

julia> 1 == 1.0
true

julia> 1 < 2
true

julia> 1.0 > 3
false

julia> 1 >= 1.0
true

julia> -1 <= 1
```

```
true

julia> -1 <= -1
true

julia> -1 <= -2
false

julia> 3 < -0.5
false
```

Integers are compared in the standard manner — by comparison of bits. Floating-point numbers are compared according to the IEEE 754 standard:

- finite numbers are ordered in the usual manner
- `Inf` is equal to itself and greater than everything else except `NaN`
- `-Inf` is equal to itself and less then everything else except `NaN`
- `NaN` is not equal to, less than, or greater than anything, including itself.

The last point is potentially suprprising and thus worth noting:

```
julia> NaN == NaN
false

julia> NaN != NaN
true

julia> NaN < NaN
false

julia> NaN > NaN
false
```

For situations where one wants to compare floating-point values so that `NaN` equals `NaN`, such as hash key comparisons, the function `isequal` is also provided, which considers `NaN`s to be equal to each other:

```
julia> isequal(NaN,NaN)
true
```

Mixed-type comparisons between signed integers, unsigned integers, and floats can be very tricky. A great deal of care has been taken to ensure that Julia does them correctly.

Unlike most languages, with the notable exception of Python, comparisons can be arbitrarily chained:

```
julia> 1 < 2 <= 2 < 3 == 3 > 2 >= 1 == 1 < 3 != 5
true
```

Chaining comparisons is often quite convenient in numerical code. Chained numeric comparisons use the `&` operator, which allows them to work on arrays. For example, `0 < A < 1` gives a boolean array whose entries are true where the corresponding elements of `A` are between 0 and 1.

Note the evaluation behavior of chained comparisons:

```
v(x) = (println(x); x)

julia> v(1) < v(2) <= v(3)
2
1
```

```
3
false
```

The middle expression is only evaluated once, rather than twice as it would be if the expression were written as `v(1) > v(2) & v(2) <= v(3)`. However, the order of evaluations in a chained comparison is undefined. It is strongly recommended not to use expressions with side effects (such as printing) in chained comparisons. If side effects are required, the short-circuit `&&` operator should be used explicitly (see *Short-Circuit Evaluation*).

1.4.3 Mathematical Functions

Julia provides a comprehensive collection of mathematical functions and operators. These mathematical operations are defined over as broad a class of numerical values as permit sensible definitions, including integers, floating-point numbers, rationals, and complexes, wherever such definitions make sense.

- `round(x)` — round `x` to the nearest integer.
- `iround(x)` — round `x` to the nearest integer, giving an integer-typed result.
- `floor(x)` — round `x` towards `-Inf`.
- `ifloor(x)` — round `x` towards `-Inf`, giving an integer-typed result.
- `ceil(x)` — round `x` towards `+Inf`.
- `iceil(x)` — round `x` towards `+Inf`, giving an integer-typed result.
- `trunc(x)` — round `x` towards zero.
- `itrunc(x)` — round `x` towards zero, giving an integer-typed result.
- `div(x,y)` — truncated division; quotient rounded towards zero.
- `fld(x,y)` — floored division; quotient rounded towards `-Inf`.
- `rem(x,y)` — remainder; satisfies `x == div(x,y)*y + rem(x,y)`, implying that sign matches `x`.
- `mod(x,y)` — modulus; satisfies `x == fld(x,y)*y + mod(x,y)`, implying that sign matches `y`.
- `gcd(x,y...)` — greatest common divisor of `x`, `y`... with sign matching `x`.
- `lcm(x,y...)` — least common multiple of `x`, `y`... with sign matching `x`.
- `abs(x)` — a positive value with the magnitude of `x`.
- `abs2(x)` — the squared magnitude of `x`.
- `sign(x)` — indicates the sign of `x`, returning -1, 0, or +1.
- `signbit(x)` — indicates whether the sign bit is on (1) or off (0).
- `copysign(x,y)` — a value with the magnitude of `x` and the sign of `y`.
- `flipsign(x,y)` — a value with the magnitude of `x` and the sign of `x*y`.
- `sqrt(x)` — the square root of `x`.
- `cbrt(x)` — the cube root of `x`.
- `hypot(x,y)` — accurate `sqrt(x^2 + y^2)` for all values of `x` and `y`.
- `exp(x)` — the natural exponential function at `x`.
- `expm1(x)` — accurate `exp(x)-1` for `x` near zero.
- `ldexp(x,n)` — `x*2^n` computed efficiently for integer values of `n`.
- `log(x)` — the natural logarithm of `x`.

- `log(b,x)` — the base `b` logarithm of `x`.
- `log2(x)` — the base 2 logarithm of `x`.
- `log10(x)` — the base 10 logarithm of `x`.
- `log1p(x)` — accurate `log(1+x)` for `x` near zero.
- `logb(x)` — returns the binary exponent of `x`.
- `erf(x)` — the error function at `x`.
- `erfc(x)` — accurate `1-erf(x)` for large `x`.
- `gamma(x)` — the gamma function at `x`.
- `lgamma(x)` — accurate `log(gamma(x))` for large `x`.

For an overview of why functions like `hypot`, `expm1`, `log1p`, and `erfc` are necessary and useful, see John D. Cook's excellent pair of blog posts on the subject: expm1, log1p, erfc, and hypot.

All the standard trigonometric functions are also defined:

```
sin    cos    tan    cot    sec    csc
sinh   cosh   tanh   coth   sech   csch
asin   acos   atan   acot   asec   acsc
acoth  asech  acsch  sinc   cosc   atan2
```

These are all single-argument functions, with the exception of atan2, which gives the angle in radians between the *x*-axis and the point specified by its arguments, interpreted as *x* and *y* coordinates. In order to compute trigonometric functions with degrees instead of radians, suffix the function with `d`. For example, `sind(x)` computes the sine of `x` where `x` is specified in degrees.

For notational convenience, the `rem` functions has an operator form:

- `x % y` is equivalent to `rem(x,y)`.

The spelled-out `rem` operator is the "canonical" form, while the `%` operator form is retained for compatibility with other systems. Like arithmetic and bitwise operators, `%` and `^` also have updating forms. As with other updating forms, `x %= y` means `x = x % y` and `x ^= y` means `x = x^y`:

```
julia> x = 2; x ^= 5; x
32

julia> x = 7; x %= 4; x
3
```

1.5 Complex and Rational Numbers

Julia ships with predefined types representing both complex and rational numbers, and supports all the mathematical operations discussed in *Mathematical Operations* on them. Promotions are defined so that operations on any combination of predefined numeric types, whether primitive or composite, behave as expected.

1.5.1 Complex Numbers

The global constant `im` is bound to the complex number *i*, representing one of the square roots of -1. It was deemed harmful to co-opt the name `i` for a global constant, since it is such a popular index variable name. Since Julia allows numeric literals to be *juxtaposed with identifiers as coefficients*, this binding suffices to provide convenient syntax for complex numbers, similar to the traditional mathematical notation:

```
julia> 1 + 2im
1 + 2im
```

You can perform all the standard arithmetic operations with complex numbers:

```
julia> (1 + 2im)*(2 - 3im)
8 + 1im

julia> (1 + 2im)/(1 - 2im)
-0.6 + 0.8im

julia> (1 + 2im) + (1 - 2im)
2 + 0im

julia> (-3 + 2im) - (5 - 1im)
-8 + 3im

julia> (-1 + 2im)^2
-3 - 4im

julia> (-1 + 2im)^2.5
2.729624464784009 - 6.9606644595719im

julia> (-1 + 2im)^(1 + 1im)
-0.27910381075826657 + 0.08708053414102428im

julia> 3(2 - 5im)
6 - 15im

julia> 3(2 - 5im)^2
-63 - 60im

julia> 3(2 - 5im)^-1.0
0.20689655172413793 + 0.5172413793103449im
```

The promotion mechanism ensures that combinations of operands of different types just work:

```
julia> 2(1 - 1im)
2 - 2im

julia> (2 + 3im) - 1
1 + 3im

julia> (1 + 2im) + 0.5
1.5 + 2.0im

julia> (2 + 3im) - 0.5im
2.0 + 2.5im

julia> 0.75(1 + 2im)
0.75 + 1.5im

julia> (2 + 3im) / 2
1.0 + 1.5im

julia> (1 - 3im) / (2 + 2im)
-0.5 - 1.0im

julia> 2im^2
```

```
-2 + 0im

julia> 1 + 3/4im
1.0 - 0.75im
```

Note that `3/4im == 3/(4*im) == -(3/4*im)`, since a literal coefficient binds more tightly than division.

Standard functions to manipulate complex values are provided:

```
julia> real(1 + 2im)
1

julia> imag(1 + 2im)
2

julia> conj(1 + 2im)
1 - 2im

julia> abs(1 + 2im)
2.23606797749979

julia> abs2(1 + 2im)
5
```

As is common, the absolute value of a complex number is its distance from zero. The `abs2` function gives the square of the absolute value, and is of particular use for complex numbers, where it avoids taking a square root. The full gamut of other mathematical functions are also defined for complex numbers:

```
julia> sqrt(im)
0.7071067811865476 + 0.7071067811865475im

julia> sqrt(1 + 2im)
1.272019649514069 + 0.7861513777574233im

julia> cos(1 + 2im)
2.0327230070196656 - 3.0518977991517997im

julia> exp(1 + 2im)
-1.1312043837568138 + 2.471726672004819im

julia> sinh(1 + 2im)
-0.48905625904129374 + 1.4031192506220407im
```

Note that mathematical functions typically return real values when applied to real numbers and complex values when applied to complex numbers. For example, `sqrt`, for example, behaves differently when applied to `-1` versus `-1 + 0im` even though `-1 == -1 + 0im`:

```
julia> sqrt(-1)
ERROR: DomainError()
 in sqrt at math.jl:111

julia> sqrt(-1 + 0im)
0.0 + 1.0im
```

If you need to construct a complex number using variables, the literal numeric coefficient notation will not work, although explicitly writing the multiplication operation will:

```
julia> a = 1; b = 2; a + b*im
1 + 2im
```

Constructing complex numbers from variable values like this, however, is not recommended. Use the `complex` function to construct a complex value directly from its real and imaginary parts instead. This construction is preferred for variable arguments because it is more efficient than the multiplication and addition construct, but also because certain values of `b` can yield unexpected results:

```
julia> complex(a,b)
1 + 2im
```

`Inf` and `NaN` propagate through complex numbers in the real and imaginary parts of a complex number as per IEEE-754 arithmetic:

```
julia> 1 + Inf*im
complex(1.0,Inf)

julia> 1 + NaN*im
complex(1.0,NaN)
```

1.5.2 Rational Numbers

Julia has a rational number type to represent exact ratios of integers. Rationals are constructed using the `//` operator:

```
julia> 2//3
2//3
```

If the numerator and denominator of a rational have common factors, they are reduced to lowest terms such that the denominator is non-negative:

```
julia> 6//9
2//3

julia> -4//8
-1//2

julia> 5//-15
-1//3

julia> -4//-12
1//3
```

This normalized form for a ratio of integers is unique, so equality of rational values can be tested by checking for equality of the numerator and denominator. The standardized numerator and denominator of a rational value can be extracted using the `num` and `den` functions:

```
julia> num(2//3)
2

julia> den(2//3)
3
```

Direct comparison of the numerator and denominator is generally not necessary, since the standard arithmetic and comparison operations are defined for rational values:

```
julia> 2//3 == 6//9
true

julia> 2//3 == 9//27
false
```

```
julia> 3//7 < 1//2
true

julia> 3//4 > 2//3
true

julia> 2//4 + 1//6
2//3

julia> 5//12 - 1//4
1//6

julia> 5//8 * 3//12
5//32

julia> 6//5 / 10//7
21//25
```

Rationals can be easily converted to floating-point numbers:

```
julia> float(3//4)
0.75
```

Conversion from rational to floating-point respects the following identity for any integral values of `a` and `b`, with the exception of the case `a == 0` and `b == 0`:

```
julia> isequal(float(a//b), a/b)
true
```

Constructing infinite rational values is acceptable:

```
julia> 5//0
Inf

julia> -3//0
-Inf

julia> typeof(ans)
Rational{Int64}
```

Trying to construct a NaN rational value, however, is not:

```
julia> 0//0
invalid rational: 0//0
```

As usual, the promotion system makes interactions with other numeric types effortless:

```
julia> 3//5 + 1
8//5

julia> 3//5 - 0.5
0.1

julia> 2//7 * (1 + 2im)
2//7 + 4//7im

julia> 2//7 * (1.5 + 2im)
0.42857142857142855 + 0.5714285714285714im

julia> 3//2 / (1 + 2im)
```

```
3//10 - 3//5im

julia> 1//2 + 2im
1//2 + 2//1im

julia> 1 + 2//3im
1//1 + 2//3im

julia> 0.5 == 1//2
true

julia> 0.33 == 1//3
false

julia> 0.33 < 1//3
true

julia> 1//3 - 0.33
0.0033333333333332993
```

1.6 Strings

Strings are finite sequences of characters. Of course, the real trouble comes when one asks what a character is. The characters that English speakers are familiar with are the letters `A`, `B`, `C`, etc., together with numerals and common punctuation symbols. These characters are standardized together with a mapping to integer values between 0 and 127 by the ASCII standard. There are, of course, many other characters used in non-English languages, including variants of the ASCII characters with accents and other modifications, related scripts such as Cyrillic and Greek, and scripts completely unrelated to ASCII and English, including Arabic, Chinese, Hebrew, Hindi, Japanese, and Korean. The Unicode standard tackles the complexities of what exactly a character is, and is generally accepted as the definitive standard addressing this problem. Depending on your needs, you can either ignore these complexities entirely and just pretend that only ASCII characters exist, or you can write code that can handle any of the characters or encodings that one may encounter when handling non-ASCII text. Julia makes dealing with plain ASCII text simple and efficient, and handling Unicode is as simple and efficient as possible. In particular, you can write C-style string code to process ASCII strings, and they will work as expected, both in terms of performance and semantics. If such code encounters non-ASCII text, it will gracefully fail with a clear error message, rather than silently introducing corrupt results. When this happens, modifying the code to handle non-ASCII data is straightforward.

There are a few noteworthy high-level features about Julia's strings:

- `String` is an abstraction, not a concrete type — many different representations can implement the `String` interface, but they can easily be used together and interact transparently. Any string type can be used in any function expecting a `String`.
- Like C and Java, but unlike most dynamic languages, Julia has a first-class type representing a single character, called `Char`. This is just a special kind of 32-bit integer whose numeric value represents a Unicode code point.
- As in Java, strings are immutable: the value of a `String` object cannot be changed. To construct a different string value, you construct a new string from parts of other strings.
- Conceptually, a string is a *partial function* from indices to characters — for some index values, no character value is returned, and instead an exception is thrown. This allows for efficient indexing into strings by the byte index of an encoded representation rather than by a character index, which cannot be implemented both efficiently and simply for variable-width encodings of Unicode strings.
- Julia supports the full range of Unicode characters: literal strings are always ASCII or UTF-8 but other encodings for strings from external sources can be supported.

1.6.1 Characters

A `Char` value represents a single character: it is just a 32-bit integer with a special literal representation and appropriate arithmetic behaviors, whose numeric value is interpreted as a Unicode code point. Here is how `Char` values are input and shown:

```
julia> 'x'
'x'

julia> typeof(ans)
Char
```

You can convert a `Char` to its integer value, i.e. code point, easily:

```
julia> int('x')
120

julia> typeof(ans)
Int64
```

On 32-bit architectures, `typeof(ans)` will be Int32. You can convert an integer value back to a `Char` just as easily:

```
julia> char(120)
'x'
```

Not all integer values are valid Unicode code points, but for performance, the `char` conversion does not check that every character value is valid. If you want to check that each converted value is a valid code point, use the `safe_char` conversion instead:

```
julia> char(0x110000)
'\U110000'

julia> safe_char(0x110000)
invalid Unicode code point: U+110000
```

As of this writing, the valid Unicode code points are `U+00` through `U+d7ff` and `U+e000` through `U+10ffff`. These have not all been assigned intelligible meanings yet, nor are they necessarily interpretable by applications, but all of these values are considered to be valid Unicode characters.

You can input any Unicode character in single quotes using `\u` followed by up to four hexadecimal digits or `\U` followed by up to eight hexadecimal digits (the longest valid value only requires six):

```
julia> '\u0'
'\0'

julia> '\u78'
'x'

julia> '\u2200'
'∀'

julia> '\U10ffff'
'\U10ffff'
```

Julia uses your system's locale and language settings to determine which characters can be printed as-is and which must be output using the generic, escaped `\u` or `\U` input forms. In addition to these Unicode escape forms, all of C's traditional escaped input forms can also be used:

```
julia> int('\0')
0
```

```
julia> int('\t')
9

julia> int('\n')
10

julia> int('\e')
27

julia> int('\x7f')
127

julia> int('\177')
127

julia> int('\xff')
255
```

You can do comparisons and a limited amount of arithmetic with `Char` values:

```
julia> 'A' < 'a'
true

julia> 'A' <= 'a' <= 'Z'
false

julia> 'A' <= 'X' <= 'Z'
true

julia> 'x' - 'a'
23

julia> 'A' + 1
'B'
```

1.6.2 String Basics

Here a variable is initialized with a simple string literal:

```
julia> str = "Hello, world.\n"
"Hello, world.\n"
```

If you want to extract a character from a string, you index into it:

```
julia> str[1]
'H'

julia> str[6]
','

julia> str[end]
'\n'
```

All indexing in Julia is 1-based: the first element of any integer-indexed object is found at index 1, and the last element is found at index `n`, when the string has a length of `n`.

In any indexing expression, the keyword `end` can be used as a shorthand for the last index (computed by `endof(str)`). You can perform arithmetic and other operations with `end`, just like a normal value:

```
julia> str[end-1]
'.'

julia> str[end/2]
' '

julia> str[end/3]
'o'

julia> str[end/4]
'l'
```

Using an index less than 1 or greater than `end` raises an error:

```
julia> str[0]
BoundsError()

julia> str[end+1]
BoundsError()
```

You can also extract a substring using range indexing:

```
julia> str[4:9]
"lo, wo"
```

Note the distinction between `str[k]` and `str[k:k]`:

```
julia> str[6]
','

julia> str[6:6]
","
```

The former is a single character value of type `Char`, while the latter is a string value that happens to contain only a single character. In Julia these are very different things.

1.6.3 Unicode and UTF-8

Julia fully supports Unicode characters and strings. As discussed above, in character literals, Unicode code points can be represented using unicode `\u` and `\U` escape sequences, as well as all the standard C escape sequences. These can likewise be used to write string literals:

```
julia> s = "\u2200 x \u2203 y"
"∀ x ∃ y"
```

Whether these Unicode characters are displayed as escapes or shown as special characters depends on your terminal's locale settings and its support for Unicode. Non-ASCII string literals are encoded using the UTF-8 encoding. UTF-8 is a variable-width encoding, meaning that not all characters are encoded in the same number of bytes. In UTF-8, ASCII characters — i.e. those with code points less than 0x80 (128) — are encoded as they are in ASCII, using a single byte, while code points 0x80 and above are encoded using multiple bytes — up to four per character. This means that not every byte index into a UTF-8 string is necessarily a valid index for a character. If you index into a string at such an invalid byte index, an error is thrown:

```
julia> s[1]
'∀'

julia> s[2]
invalid UTF-8 character index
```

```
julia> s[3]
invalid UTF-8 character index

julia> s[4]
' '
```

In this case, the character ∀ is a three-byte character, so the indices 2 and 3 are invalid and the next character's index is 4.

Because of variable-length encodings, the number of character in a string (given by `length(s)`) is not always the same as the last index. If you iterate through the indices 1 through `endof(s)` and index into `s`, the sequence of characters returned, when errors aren't thrown, is the sequence of characters comprising the string `s`. Thus, we do have the identity that `length(s) <= endof(s)` since each character in a string must have its own index. The following is an inefficient and verbose way to iterate through the characters of `s`:

```
julia> for i = 1:endof(s)
         try
           println(s[i])
         catch
           # ignore the index error
         end
       end
∀

x

∃

y
```

The blank lines actually have spaces on them. Fortunately, the above awkward idiom is unnecessary for iterating through the characters in a string, since you can just use the string as an iterable object, no exception handling required:

```
julia> for c in s
         println(c)
       end
∀

x

∃

y
```

UTF-8 is not the only encoding that Julia supports, and adding support for new encodings is quite easy, but discussion of other encodings and how to implement support for them is beyond the scope of this document for the time being. For further discussion of UTF-8 encoding issues, see the section below on byte array literals, which goes into some greater detail.

1.6.4 Interpolation

One of the most common and useful string operations is concatenation:

```
julia> greet = "Hello"
"Hello"

julia> whom = "world"
```

```
"world"

julia> string(greet, ", ", whom, ".\n")
"Hello, world.\n"
```

Constructing strings like this can become a bit cumbersome, however. To reduce the need for these verbose calls to `string`, Julia allows interpolation into string literals using `$`, as in Perl:

```
julia> "$greet, $whom.\n"
"Hello, world.\n"
```

This is more readable and convenient and equivalent to the above string concatenation — the system rewrites this apparent single string literal into a concatenation of string literals with variables.

The shortest complete expression after the `$` is taken as the expression whose value is to be interpolated into the string. Thus, you can interpolate any expression into a string using parentheses:

```
julia> "1 + 2 = $(1 + 2)"
"1 + 2 = 3"
```

Both concatenation and string interpolation call the generic `string` function to convert objects into `String` form. Most non-`String` objects are converted to strings as they are shown in interactive sessions:

```
julia> v = [1,2,3]
3-element Int64 Array:
 1
 2
 3

julia> "v: $v"
"v: [1, 2, 3]"
```

The `string` function is the identity for `String` and `Char` values, so these are interpolated into strings as themselves, unquoted and unescaped:

```
julia> c = 'x'
'x'

julia> "hi, $c"
"hi, x"
```

To include a literal `$` in a string literal, escape it with a backslash:

```
julia> print("I have \$100 in my account.\n")
I have $100 in my account.
```

1.6.5 Common Operations

You can lexicographically compare strings using the standard comparison operators:

```
julia> "abracadabra" < "xylophone"
true

julia> "abracadabra" == "xylophone"
false

julia> "Hello, world." != "Goodbye, world."
true
```

```
julia> "1 + 2 = 3" == "1 + 2 = $(1 + 2)"
true
```

You can search for the index of a particular character using the `strchr` function:

```
julia> strchr("xylophone", 'x')
1

julia> strchr("xylophone", 'p')
5

julia> strchr("xylophone", 'z')
0
```

You can start the search for a character at a given offset by providing a third argument:

```
julia> strchr("xylophone", 'o')
4

julia> strchr("xylophone", 'o', 5)
7

julia> strchr("xylophone", 'o', 8)
0
```

Another handy string function is `repeat`:

```
julia> repeat(".:Z:.", 10)
".:Z:..:Z:..:Z:..:Z:..:Z:..:Z:..:Z:..:Z:..:Z:..:Z:."
```

Some other useful functions include:

- `endof(str)` gives the maximal (byte) index that can be used to index into `str`.
- `length(str)` the number of characters in `str`.
- `i = start(str)` gives the first valid index at which a character can be found in `str` (typically 1).
- `c, j = next(str,i)` returns next character at or after the index `i` and the next valid character index following that. With `start` and `endof`, can be used to iterate through the characters in `str`.
- `ind2chr(str,i)` gives the number of characters in `str` up to and including any at index `i`.
- `chr2ind(str,j)` gives the index at which the `j`th character in `str` occurs.

1.6.6 Non-Standard String Literals

There are situations when you want to construct a string or use string semantics, but the behavior of the standard string construct is not quite what is needed. For these kinds of situations, Julia provides *non-standard string literals*. A non-standard string literal looks like a regular double-quoted string literal, but is immediately prefixed by an identifier, and doesn't behave quite like a normal string literal.

1.6.7 Regular Expressions

Julia has Perl-compatible regular expressions, as provided by the PCRE library. Regular expressions are related to strings in two ways: the obvious connection is that regular expressions are used to find regular patterns in strings; the other connection is that regular expressions are themselves input as strings, which are parsed into a state machine that can be used to efficiently search for patterns in strings. In Julia, regular expressions are input using non-standard

string literals prefixed with various identifiers beginning with `r`. The most basic regular expression literal without any options turned on just uses `r"..."`:

```
julia> r"^\s*(?:#|$)"
r"^\s*(?:#|$)"

julia> typeof(ans)
Regex
```

To check if a regex matches a string, use the `ismatch` function:

```
julia> ismatch(r"^\s*(?:#|$)", "not a comment")
false

julia> ismatch(r"^\s*(?:#|$)", "# a comment")
true
```

As one can see here, `ismatch` simply returns true or false, indicating whether the given regex matches the string or not. Commonly, however, one wants to know not just whether a string matched, but also *how* it matched. To capture this information about a match, use the `match` function instead:

```
julia> match(r"^\s*(?:#|$)", "not a comment")

julia> match(r"^\s*(?:#|$)", "# a comment")
RegexMatch("#")
```

If the regular expression does not match the given string, `match` returns `nothing` — a special value that does not print anything at the interactive prompt. Other than not printing, it is a completely normal value and you can test for it programmatically:

```
m = match(r"^\s*(?:#|$)", line)
if m == nothing
  println("not a comment")
else
  println("blank or comment")
end
```

If a regular expression does match, the value returned by `match` is a `RegexMatch` object. These objects record how the expression matches, including the substring that the pattern matches and any captured substrings, if there are any. This example only captures the portion of the substring that matches, but perhaps we want to capture any non-blank text after the comment character. We could do the following:

```
julia> m = match(r"^\s*(?:#\s*(.*?)\s*$|$)", "# a comment ")
RegexMatch("# a comment ", 1="a comment")
```

You can extract the following info from a `RegexMatch` object:

- the entire substring matched: `m.match`
- the captured substrings as a tuple of strings: `m.captures`
- the offset at which the whole match begins: `m.offset`
- the offsets of the captured substrings as a vector: `m.offsets`

For when a capture doesn't match, instead of a substring, `m.captures` contains `nothing` in that position, and `m.offsets` has a zero offset (recall that indices in Julia are 1-based, so a zero offset into a string is invalid). Here's is a pair of somewhat contrived examples:

```
julia> m = match(r"(a|b)(c)?(d)", "acd")
RegexMatch("acd", 1="a", 2="c", 3="d")
```

```
julia> m.match
"acd"

julia> m.captures
3-element Union(UTF8String,ASCIIString,Nothing) Array:
 "a"
 "c"
 "d"

julia> m.offset
1

julia> m.offsets
3-element Int64 Array:
 1
 2
 3

julia> m = match(r"(a|b)(c)?(d)", "ad")
RegexMatch("ad", 1="a", 2=nothing, 3="d")

julia> m.match
"ad"

julia> m.captures
3-element Union(UTF8String,ASCIIString,Nothing) Array:
 "a"
 nothing
 "d"

julia> m.offset
1

julia> m.offsets
3-element Int64 Array:
 1
 0
 2
```

It is convenient to have captures returned as a tuple so that one can use tuple destructuring syntax to bind them to local variables:

```
julia> first, second, third = m.captures; first
"a"
```

You can modify the behavior of regular expressions by some combination of the flags `i`, `m`, `s`, and `x` after the closing double quote mark. These flags have the same meaning as they do in Perl, as explained in this excerpt from the perlre manpage:

```
i   Do case-insensitive pattern matching.

    If locale matching rules are in effect, the case map is taken
    from the current locale for code points less than 255, and
    from Unicode rules for larger code points. However, matches
    that would cross the Unicode rules/non-Unicode rules boundary
    (ords 255/256) will not succeed.

m   Treat string as multiple lines.  That is, change "^" and "$"
```

```
    from matching the start or end of the string to matching the
    start or end of any line anywhere within the string.

s   Treat string as single line.  That is, change "." to match any
    character whatsoever, even a newline, which normally it would
    not match.

    Used together, as r""ms, they let the "." match any character
    whatsoever, while still allowing "^" and "$" to match,
    respectively, just after and just before newlines within the
    string.

x   Tells the regular expression parser to ignore most whitespace
    that is neither backslashed nor within a character class. You
    can use this to break up your regular expression into
    (slightly) more readable parts. The '#' character is also
    treated as a metacharacter introducing a comment, just as in
    ordinary code.
```

For example, the following regex has all three flags turned on:

```
julia> r"a+.*b+.*?d$"ism
r"a+.*b+.*?d$"ims

julia> match(r"a+.*b+.*?d$"ism, "Goodbye,\nOh, angry,\nBad world\n")
RegexMatch("angry,\nBad world")
```

Byte Array Literals

Another useful non-standard string literal is the byte-array string literal: `b"..."`. This form lets you use string notation to express literal byte arrays — i.e. arrays of `Uint8` values. The convention is that non-standard literals with uppercase prefixes produce actual string objects, while those with lowercase prefixes produce non-string objects like byte arrays or compiled regular expressions. The rules for byte array literals are the following:

- ASCII characters and ASCII escapes produce a single byte.
- `\x` and octal escape sequences produce the *byte* corresponding to the escape value.
- Unicode escape sequences produce a sequence of bytes encoding that code point in UTF-8.

There is some overlap between these rules since the behavior of `\x` and octal escapes less than 0x80 (128) are covered by both of the first two rules, but here these rules agree. Together, these rules allow one to easily use ASCII characters, arbitrary byte values, and UTF-8 sequences to produce arrays of bytes. Here is an example using all three:

```
julia> b"DATA\xff\u2200"
[68,65,84,65,255,226,136,128]
```

The ASCII string "DATA" corresponds to the bytes 68, 65, 84, 65. `\xff` produces the single byte 255. The Unicode escape `\u2200` is encoded in UTF-8 as the three bytes 226, 136, 128. Note that the resulting byte array does not correspond to a valid UTF-8 string — if you try to use this as a regular string literal, you will get a syntax error:

```
julia> "DATA\xff\u2200"
syntax error: invalid UTF-8 sequence
```

Also observe the significant distinction between `\xff` and `\uff`: the former escape sequence encodes the *byte 255*, whereas the latter escape sequence represents the *code point 255*, which is encoded as two bytes in UTF-8:

```
julia> b"\xff"
1-element Uint8 Array:
 0xff

julia> b"\uff"
2-element Uint8 Array:
 0xc3
 0xbf
```

In character literals, this distinction is glossed over and `\xff` is allowed to represent the code point 255, because characters *always* represent code points. In strings, however, `\x` escapes always represent bytes, not code points, whereas `\u` and `\U` escapes always represent code points, which are encoded in one or more bytes. For code points less than `\u80`, it happens that the the UTF-8 encoding of each code point is just the single byte produced by the corresponding `\x` escape, so the distinction can safely be ignored. For the escapes `\x80` through `\xff` as compared to `\u80` through `\uff`, however, there is a major difference: the former escapes all encode single bytes, which — unless followed by very specific continuation bytes — do not form valid UTF-8 data, whereas the latter escapes all represent Unicode code points with two-byte encodings.

If this is all extremely confusing, try reading "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets". It's an excellent introduction to Unicode and UTF-8, and may help alleviate some confusion regarding the matter.

1.7 Funções

Em Julia, uma função é um objeto que mapeia uma tupla de valores, os argumentos, a um valor de retorno. As funções, em Julia, são diferentes das funções matemáticas, pois as funções podem se alterar e afetadas pelo estado global do programa. A sintaxe básica para definir uma funções em Julia é:

```
function f(x,y)
  x + y
end
```

Esta sintaxe é similar a do MATLAB, mas há algumas diferenças significativas:

- No MATLAB, esta definição deve ser salvar em um arquivo, nomeado `f.m`, enquanto que que em Julia, esta declaração pode aparecer em qualquer lugar, incluindo em uma sessão interativa.
- No MATLAB, a declaração `end` final é opcional, sendo implicado pelo fim do arquivo. Em Julia, essa declaração `end` é obrigatória.
- No MATLAB, esta função irá imprimir o valor `x + y` mas não retornará nenhum valor, enquanto que em Julia, a última expressão avaliada é o valor de retorno da função.
- Os valores de uma expressão nunca são mostrados automaticamente exceto em sessões interativas. Ponto-e-vírgula são exigidos somente para separar expressões na mesma linha.

Geralmente, enquanto a sintaxe da definição de função é remanescente do MATLAB, a similaridade é apenas superficial. Logo, ao invés de continuar comparando as duas, a seguir, nós simplesmente descreveremos o comportamento das funções em Julia.

Existe uma forma mais compacta de definir uma função em Julia. A sintaxe tradicional de declaração de função apresentada acima é equivalente a forma compactada a seguir:

```
f(x,y) = x + y
```

Nessa forma compacta, o corpo da função deve ser uma única expressão, embora possa ser uma expressão composta (veja *Compound Expressions*). Definições de funções de forma curta e simples são comuns em Julia. A sintaxe curta da função é bastante idiomática, reduzindo consideravelmente a digitação e a poluição visual.

Uma função é chamada usando a sintaxe tradicional de parêntese:

```
julia> f(2,3)
5
```

Sem parênteses, a expressão `f` refere-se ao objeto da função, e pode ser passada como qualquer valor:

```
julia> g = f;

julia> g(2,3)
5
```

Há outras duas maneiras que as funções podem ser aplicadas: usando operadores com sintaxe especial para certos nomes de funções (veja Operadores são Funções), ou com a função `apply`:

```
julia> apply(f,2,3)
5
```

A função `apply` aplicam seu primeiro argumento - um objeto de função - a seus argumentos restantes.

1.7.1 A declaração "return"

O valor retornado por uma função é o valor da última expressão avaliada, que, por padrão é a última expressão no corpo da definição da função. Na função de exemplo, `f`, da seção anterior isto é o valor da expressão `x + y`. Como em C e na maioria das outras línguas imperativas ou funcionais, a declaração `return` faz com que uma função retorne imediatamente, fornecendo uma expressão cujo o valor será retornado:

```
function g(x,y)
  return x * y
  x + y
end
```

Como definições de funções podem ser feitas em sessões interativas, é fácil comparar estas definições:

```
f(x,y) = x + y

function g(x,y)
  return x * y
  x + y
end

julia> f(2,3)
5

julia> g(2,3)
6
```

Naturalmente, em uma função cujo corpo é linear como `g`, o uso do `return` é injustificado pois a expressão `x + y` nunca é avaliada e nós poderíamos simplesmente tornar `x * y` a última expressão na função e omitir `return`. Já em conjunto com outras declarações de controle deo fluxo, contudo, o `return` é do uso real. A seguir, por exemplo, está uma funciona que calcula o comprimento da hipotenusa de um triângulo retângulo com lados de comprimento *x* e *y*, evitando *overflow*:

```
function hypot(x,y)
  x = abs(x)
  y = abs(y)
  if x > y
    r = y/x
    return x*sqrt(1+r*r)
```

```
  end
  if y == 0
    return zero(x)
  end
  r = x/y
  return y*sqrt(1+r*r)
end
```

Há três possíveis pontos de retorno nesta função, retornando os valores de três expressões diferentes, dependendo dos valores de *x* e *y*. O `return` na última linha podia ser omitido pois ele é o último expressão.

1.7.2 Operadores são funções

Em Julia, a maioria dos operadores são apenas funções com suport para sintaxe especial. As exceções são operadores com semântica especial como o `&&` e `||`. Estes operadores não podem ser funções pois o *short-circuit evaluation* (veja *Short-Circuit Evaluation*) exige que seus operandos não sejam avaliados antes da avaliação do operador. Logo, você também pode aplicá-los usam uma lista de argumento entre parênteses, de forma semelhante como qualquer outra função:

```
julia> 1 + 2 + 3
6

julia> +(1,2,3)
6
```

A forma infixa é exatamente equivalente a forma padrão - na verdade a primeira forma é convertida para uma chamada de função internamente. Isto significa que você também pode atribuir e passar operadores como + e * da mesma forma como você faria para outra função:

```
julia> f = +;

julia> f(1,2,3)
6
```

Sob o nome `f`, a função suporta a forma infixa.

1.7.3 Funções Anônimas

Funções em Julia são objetos de primeira classe: podem ser atribuídos a variáveis, chamadas usando a sintaxe padrão para chamada de função a partir da variável que foram atribuídas. Podem ser usadas como argumentos, e podem ser retornadas como valores. Também pode ser criadas anonimamente, sem ter um nome:

```
julia> x -> x^2 + 2x - 1
#<function>
```

Isto cria uma função sem nome que possue um argumento e que retorna o valor do polinômio *x* ^2 + 2 *x* - 1. O uso principal para funções anônimas é serem passadas para funções que recebem outras funções como argumentos. Um exemplo clássico é a função do `map`, que aplica uma função a cada valor de um vetor e retorna um novo vetor que contem os valores resultantes:

```
julia> map(round, [1.2, 3.5, 1.7])
3-element Float64 Array:
 1.0
 4.0
 2.0
```

Não existe problema se uma função, já nomeada, que efetua a transformação desejada já existe para ser passada como o primeiro argumento da função `map`. Entretanto, frequentemente, não existe a função desejada pronta para uso. Nestas situações, a função anónima permite a criação de um objeto função para um único uso sem precisar atribuir um nome:

```
julia> map(x -> x^2 + 2x - 1, [1,3,-1])
3-element Int64 Array:
 2
 14
 -2
```

Uma função anónima que aceita mais de um argumentos pode ser escrita usando a sintaxe `(x,y,z)->2x+y-z`. Uma função anónima sem argumento é escrita como `()->3`. A ideia de uma função sem argumentos pode parecer estranha, mas é útil para "atrasar" algum cálculo. Neste uso, um bloco de código é envolvido em uma função sem argumento, que é posteriormente invocada chamando `f()`.

1.7.4 Retornando mais de um valor

Em Julia, uma tupla deve ser retornada para simular o retorno de mais de um valor. Contudo, os tuplas podem ser criadas e destruidas sem precisar de parênteses, fornecendo a ilusão de que mais de um valor esta sendo retornado, ao invés de uma única tuple. Por exemplo, a função a seguir retorna um par de valores:

```
function foo(a,b)
  a+b, a*b
end
```

Se você chama essa função em uma sessão interativa sem atribuir o valor de retorno em nenhum lugar, você verá a tupla sendo retornada:

```
julia> foo(2,3)
(5,6)
```

Um uso típico de funções que retornam mais de um valor, contudo, extrai cada valor em uma variável. Julia suporta a "destruição" simplificada de tuplas que facilitam isto:

```
julia> x, y = foo(2,3);

julia> x
5

julia> y
6
```

Você também pode retornar mais de um valores através do uso explícito da expressão‘‘return‘‘:

```
function foo(a,b)
  return a+b, a*b
end
```

Isto tem exatamente mesmo efeito que a definição anterior de `foo`.

1.7.5 Funções com Número Variado de Argumentos

Frequentemente, é conveniente poder escrever funções que tomam um número arbitrário de argumentos. Tais funções são tradicional conhecidas como funções *varargs*, que um acrônimo para "variable number of arguments" (ou "número variável de argumentos", em tradução literal). Você pode definir uma função *varargs* utilizando depois do último argumento uma elipse (`...`):

```
bar(a,b,x...) = (a,b,x)
```

As variávies `a` e `b` são atribuidas aos primeiros dois argumento como é o costume, e a variável `x` é atribuida para coleção de zero ou mais valores passados para `bar` depois dos seus primeiros dois argumentos:

```
julia> bar(1,2)
(1,2,())

julia> bar(1,2,3)
(1,2,(3,))

julia> bar(1,2,3,4)
(1,2,(3,4))

julia> bar(1,2,3,4,5,6)
(1,2,(3,4,5,6))
```

Em todos estes casos, `x` corresponde a uma tupla dos valores passado a `bar`.

Por outros lado, é frequentemente necessário "dividir" os valores presentes em uma coleção iterável em argumentos individuais para uma chamda de função. Para fazer isso, usa-se de forma análoga `...` mas na chamada da função:

```
julia> x = (3,4)
(3,4)

julia> bar(1,2,x...)
(1,2,(3,4))
```

Neste caso uma tupla de valores é dividido na chamada de uma função *varargs* precisamente onde o número de argumentos variável vai. Isso não precisar necessidade ser o caso:

```
julia> x = (2,3,4)
(2,3,4)

julia> bar(1,x...)
(1,2,(3,4))

julia> x = (1,2,3,4)
(1,2,3,4)

julia> bar(x...)
(1,2,(3,4))
```

Além disso, não é necessário dividir uma tupla para passá-la para uma função:

```
julia> x = [3,4]
2-element Int64 Array:
 3
 4

julia> bar(1,2,x...)
(1,2,(3,4))

julia> x = [1,2,3,4]
4-element Int64 Array:
 1
 2
 3
 4
```

```
julia> bar(x...)
(1,2,(3,4))
```

Além disso, a função não precisa ser *varargs* para que os argumentos sejam divididos (embora é frequentemente):

```
baz(a,b) = a + b

julia> args = [1,2]
2-element Int64 Array:
 1
 2

julia> baz(args...)
3

julia> args = [1,2,3]
3-element Int64 Array:
 1
 2
 3

julia> baz(args...)
no method baz(Int64,Int64,Int64)
```

Como você pode ver, se o objeto a ser dividido na chamada da função resultar em um número de argumentos diferente do esperado, a função irá falhar, de forma semelhante se um muitos argumentos tivessem sido passados de forma explícita.

1.7.6 Argumentos opcionais

Em muitos casos, os argumentos de uma função possuem valores padrões que não precisam ser passados explicitamentes em toda chamada de função. Por exemplo, a função `parseint(num,base)` interpreta interpreta uma *string* como um número em alguma base. O valor padrão para o argumento `base` é `10`. Este comportamento pode ser expresso como:

```
function parseint(num, base=10)
    ###
end
```

Com esta definição, a função pode ser chamada com um ou dois argumentos, e `10` é passado automaticamente quando um segundo argumento não é especificado:

```
julia> parseint("12",10)
12

julia> parseint("12",3)
5

julia> parseint("12")
12
```

Argumentos opcionais são na verdade apenas uma sintaxe conveniente para escrever mais de uma definição para um método com números diferentes de argumentos (veja *Methods*).

1.7.7 Argumento nomeado

Algumas funções precisam de um grande número de argumentos, ou têm um grande número de comportamentos. Recordar como chamar tais funções pode ser difícil. Argumentos nomeados, ou *keyword arguments*, podem facilitar o uso destas funções complexas e estendida ao permitindo que os argumentos sejam identificados por nome em vez de apenas pela da posição.

Por exemplo, considere uma função `plot` que traça uma linha. Esta função deve ter muitas opções, para controlar o estilo, largura, cor, ... da linha. Se ela aceitar argumentos nomeados, um possível a chamada pode parecer com `plot(x, y, width=2)`, onde escolhemos especificar somente a largura da linha. Observe que isto serve para duas finalidades. A chamada da função é mais fácil de ler, desde que podemos etiquetar os argumentos com seu significado. E também, torna-se possível passar qualquer subconjunto de argumentos em qualquer ordem.

As funções com argumentos nomeados são definidas usando um ponto-e-vírgula na declaração:

```
function plot(x, y; style="solid", width=1, color="black")
    ###
end
```

Argumentos nomeados adicionais podem ser informados utilizando `...`, como nas funções *vargargs*:

```
function f(x; args...)
    ###
end
```

Dentro de `f`, `args` será uma coleção de tuplas do tipo `(chave,valor)`, onde cada `chave` é um símbolo. Tais coleções podem ser passadas como argumentos nomeados usando um ponto-e-vírgula na chamada da função, `f(x; k.....)`. Dicionários podem ser usados para esta finalidade.

1.7.8 Sintaxe de bloco para argumentos de função

Passar funções como argumentos a outras funções é uma técnica poderosa, mas a sintaxe para isso não é sempre conveniente. Tais chamadas são especialmente difíceis de escrever quando o argumento da função exige mais de uma linhas. Por um exemplo, considere a chamada da função `map` passando uma função em diversos casos:

```
map(x->begin
           if x < 0 && iseven(x)
               return 0
           elseif x == 0
               return 1
           else
               return x
           end
       end,
    [A, B, C])
```

Julia possue uma palavra reservado `do` para reescrevendo este código de forma mais clara:

```
map([A, B, C]) do x
    if x < 0 && iseven(x)
        return 0
    elseif x == 0
        return 1
    else
        return x
    end
end
```

A sintaxe `do x` cria uma função anónima com o argumento `x` e passa essa função como o primeiro argumento de `mapa`. Esta sintaxe facilita usar funções para estender a línguagem, pois as chamadas parecem com blocos de código convencional. Há muitos usos diferentes da função `mapa`, como gerenciar o estado do sistema. Por exemplo, a biblioteca padrão fornece uma função `cd` para rodar código em um diretório especificado, e retornar ao diretório anterior quando o código terminar ou abortar. Existe também uma função `open` que roda código garantindo que o arquivo aberto será eventualmente fechado. Podemos combinar estas funções para escrever com segurança um arquivo em um determinado diretório:

```
cd("data") do
    open("outfile", "w") do f
        write(f, data)
    end
end
```

O argumento da função `cd` não recebe nenhum argumento; é apenas um bloco de código. O argumento da função `open` recebe informações de como lidar com o arquivo aberto.

1.7.9 Leitura adicional

Devemos mencionar aqui que esta não é uma imagem completa sobre definições de funções. Julia tem um sofisticado sistema de tipos e permite mais de uma declarações baseada no tipo de argumentos. Nenhuns dos exemplos dados aqui fornecem qualquer tipo de anotações sobre seus argumentos, significando que são aplicáveis a todos os tipos de argumentos. O sistema de tipos é descrito em *Types* e a definição de funções em termos de métodos escolhidos com base no tipo dos argumentos em tempo de execução é descrito em :ref: *man-methods*.

1.8 Control Flow

Julia provides a variety of control flow constructs:

- *Compound Expressions*: `begin` and `(;)`.
- *Conditional Evaluation*: `if-elseif-else` and `?:` (ternary operator).
- *Short-Circuit Evaluation*: `&&`, `||` and chained comparisons.
- *Repeated Evaluation: Loops*: `while` and `for`.
- *Exception Handling*: `try-catch`, `error` and `throw`.
- *Tasks (aka Coroutines)*: `yieldto`.

The first five control flow mechanisms are standard to high-level programming languages. Tasks are not so standard: they provide non-local control flow, making it possible to switch between temporarily-suspended computations. This is a powerful construct: both exception handling and cooperative multitasking are implemented in Julia using tasks. Everyday programming requires no direct usage of tasks, but certain problems can be solved much more easily by using tasks.

1.8.1 Compound Expressions

Sometimes it is convenient to have a single expression which evaluates several subexpressions in order, returning the value of the last subexpression as its value. There are two Julia constructs that accomplish this: `begin` blocks and `(;)` chains. The value of both compound expression constructs is that of the last subexpression. Here's an example of a `begin` block:

```
julia> z = begin
         x = 1
         y = 2
         x + y
       end
3
```

Since these are fairly small, simple expressions, they could easily be placed onto a single line, which is where the `(;)` chain syntax comes in handy:

```
julia> z = (x = 1; y = 2; x + y)
3
```

This syntax is particularly useful with the terse single-line function definition form introduced in *Funções*. Although it is typical, there is no requirement that `begin` blocks be multiline or that `(;)` chains be single-line:

```
julia> begin x = 1; y = 2; x + y end
3

julia> (x = 1;
        y = 2;
        x + y)
3
```

1.8.2 Conditional Evaluation

Conditional evaluation allows portions of code to be evaluated or not evaluated depending on the value of a boolean expression. Here is the anatomy of the `if-elseif-else` conditional syntax:

```
if x < y
  println("x is less than y")
elseif x > y
  println("x is greater than y")
else
  println("x is equal to y")
end
```

The semantics are just what you'd expect: if the condition expression `x < y` is `true`, then the corresponding block is evaluated; otherwise the condition expression `x > y` is evaluated, and if it is `true`, the corresponding block is evaluated; if neither expression is true, the `else` block is evaluated. Here it is in action:

```
julia> function test(x, y)
         if x < y
           println("x is less than y")
         elseif x > y
           println("x is greater than y")
         else
           println("x is equal to y")
         end
       end

julia> test(1, 2)
x is less than y

julia> test(2, 1)
x is greater than y
```

```
julia> test(1, 1)
x is equal to y
```

The `elseif` and `else` blocks are optional, and as many `elseif` blocks as desired can be used. The condition expressions in the `if-elseif-else` construct are evaluated until the first one evaluates to `true`, after which the associated block is evaluated, and no further condition expressions or blocks are evaluated.

Unlike C, MATLAB, Perl, Python, and Ruby — but like Java, and a few other stricter, typed languages — it is an error if the value of a conditional expression is anything but `true` or `false`:

```
julia> if 1
         println("true")
       end
type error: lambda: in if, expected Bool, got Int64
```

This error indicates that the conditional was of the wrong type: `Int64` rather than the required `Bool`.

The so-called "ternary operator", `?:`, is closely related to the `if-elseif-else` syntax, but is used where a conditional choice between single expression values is required, as opposed to conditional execution of longer blocks of code. It gets its name from being the only operator in most languages taking three operands:

```
a ? b : c
```

The expression `a`, before the `?`, is a condition expression, and the ternary operation evaluates the expression `b`, before the `:`, if the condition `a` is `true` or the expression `c`, after the `:`, if it is `false`.

The easiest way to understand this behavior is to see an example. In the previous example, the `println` call is shared by all three branches: the only real choice is which literal string to print. This could be written more concisely using the ternary operator. For the sake of clarity, let's try a two-way version first:

```
julia> x = 1; y = 2;

julia> println(x < y ? "less than" : "not less than")
less than

julia> x = 1; y = 0;

julia> println(x < y ? "less than" : "not less than")
not less than
```

If the expression `x < y` is true, the entire ternary operator expression evaluates to the string `"less than"` and otherwise it evaluates to the string `"not less than"`. The original three-way example requires chaining multiple uses of the ternary operator together:

```
julia> test(x, y) = println(x < y ? "x is less than y"    :
                            x > y ? "x is greater than y" : "x is equal to y")

julia> test(1, 2)
x is less than y

julia> test(2, 1)
x is greater than y

julia> test(1, 1)
x is equal to y
```

To facilitate chaining, the operator associates from right to left.

It is significant that like `if-elseif-else`, the expressions before and after the `:` are only evaluated if the condition expression evaluates to `true` or `false`, respectively:

```
v(x) = (println(x); x)

julia> 1 < 2 ? v("yes") : v("no")
yes
"yes"

julia> 1 > 2 ? v("yes") : v("no")
no
"no"
```

1.8.3 Short-Circuit Evaluation

Short-circuit evaluation is quite similar to conditional evaluation. The behavior is found in most imperative programming languages having the `&&` and `||` boolean operators: in a series of boolean expressions connected by these operators, only the minimum number of expressions are evaluated as are necessary to determine the final boolean value of the entire chain. Explicitly, this means that:

- In the expression `a && b`, the subexpression `b` is only evaluated if `a` evaluates to `true`.
- In the expression `a || b`, the subexpression `b` is only evaluated if `a` evaluates to `false`.

The reasoning is that `a && b` must be `false` if `a` is `false`, regardless of the value of `b`, and likewise, the value of `a || b` must be true if `a` is `true`, regardless of the value of `b`. Both `&&` and `||` associate to the right, but `&&` has higher precedence than than `||` does. It's easy to experiment with this behavior:

```
t(x) = (println(x); true)
f(x) = (println(x); false)

julia> t(1) && t(2)
1
2
true

julia> t(1) && f(2)
1
2
false

julia> f(1) && t(2)
1
false

julia> f(1) && f(2)
1
false

julia> t(1) || t(2)
1
true

julia> t(1) || f(2)
1
true

julia> f(1) || t(2)
1
2
true
```

```
julia> f(1) || f(2)
1
2
false
```

You can easily experiment in the same way with the associativity and precedence of various combinations of `&&` and `||` operators.

If you want to perform boolean operations *without* short-circuit evaluation behavior, you can use the bitwise boolean operators introduced in *Mathematical Operations*: `&` and `|`. These are normal functions, which happen to support infix operator syntax, but always evaluate their arguments:

```
julia> f(1) & t(2)
1
2
false

julia> t(1) | t(2)
1
2
true
```

Just like condition expressions used in `if`, `elseif` or the ternary operator, the operands of `&&` or `||` must be boolean values (`true` or `false`). Using a non-boolean value is an error:

```
julia> 1 && 2
type error: lambda: in if, expected Bool, got Int64
```

1.8.4 Repeated Evaluation: Loops

There are two constructs for repeated evaluation of expressions: the `while` loop and the `for` loop. Here is an example of a `while` loop:

```
julia> i = 1;

julia> while i <= 5
         println(i)
         i += 1
       end
1
2
3
4
5
```

The `while` loop evaluates the condition expression (`i < n` in this case), and as long it remains `true`, keeps also evaluating the body of the `while` loop. If the condition expression is `false` when the `while` loop is first reached, the body is never evaluated.

The `for` loop makes common repeated evaluation idioms easier to write. Since counting up and down like the above `while` loop does is so common, it can be expressed more concisely with a `for` loop:

```
julia> for i = 1:5
         println(i)
       end
1
2
3
```

```
4
5
```

Here the `1:5` is a `Range` object, representing the sequence of numbers 1, 2, 3, 4, 5. The `for` loop iterates through these values, assigning each one in turn to the variable `i`. One rather important distinction between the previous `while` loop form and the `for` loop form is the scope during which the variable is visible. If the variable `i` has not been introduced in an other scope, in the `for` loop form, it is visible only inside of the `for` loop, and not afterwards. You'll either need a new interactive session instance or a different variable name to test this:

```
julia> for j = 1:5
         println(j)
       end
1
2
3
4
5

julia> j
j not defined
```

See *Variables and Scoping* for a detailed explanation of variable scope and how it works in Julia.

In general, the `for` loop construct can iterate over any container. In these cases, the alternative (but fully equivalent) keyword `in` is typically used instead of =, since it makes the code read more clearly:

```
julia> for i in [1,4,0]
         println(i)
       end
1
4
0

julia> for s in ["foo","bar","baz"]
         println(s)
       end
foo
bar
baz
```

Various types of iterable containers will be introduced and discussed in later sections of the manual (see, e.g., *Arrays*).

It is sometimes convenient to terminate the repetition of a `while` before the test condition is falsified or stop iterating in a `for` loop before the end of the iterable object is reached. This can be accomplished with the `break` keyword:

```
julia> i = 1;

julia> while true
         println(i)
         if i >= 5
           break
         end
         i += 1
       end
1
2
3
4
5
```

```
julia> for i = 1:1000
         println(i)
         if i >= 5
           break
         end
       end
1
2
3
4
5
```

The above `while` loop would never terminate on its own, and the `for` loop would iterate up to 1000. These loops are both exited early by using the `break` keyword.

In other circumstances, it is handy to be able to stop an iteration and move on to the next one immediately. The `continue` keyword accomplishes this:

```
julia> for i = 1:10
         if i % 3 != 0
           continue
         end
         println(i)
       end
3
6
9
```

This is a somewhat contrived example since we could produce the same behavior more clearly by negating the condition and placing the `println` call inside the `if` block. In realistic usage there is more code to be evaluated after the `continue`, and often there are multiple points from which one calls `continue`.

Multiple nested `for` loops can be combined into a single outer loop, forming the cartesian product of its iterables:

```
julia> for i = 1:2, j = 3:4
         println((i, j))
       end
(1,3)
(1,4)
(2,3)
(2,4)
```

1.8.5 Exception Handling

When an unexpected condition occurs, a function may be unable to return a reasonable value to its caller. In such cases, it may be best for the exceptional condition to either terminate the program, printing a diagnostic error message, or if the programmer has provided code to handle such exceptional circumstances, allow that code to take the appropriate action.

The `error` function is used to indicate that an unexpected condition has occurred which should interrupt the normal flow of control. The built in `sqrt` function returns `DomainError()` if applied to a negative real value:

```
julia> sqrt(-1)
DomainError()
```

Suppose we want to stop execution immediately if the square root of a negative number is taken. To do this, we can define a fussy version of the `sqrt` function that raises an error if its argument is negative:

```
fussy_sqrt(x) = x >= 0 ? sqrt(x) : error("negative x not allowed")

julia> fussy_sqrt(2)
1.4142135623730951

julia> fussy_sqrt(-1)
negative x not allowed
```

If `fussy_sqrt` is called with a negative value from another function, instead of trying to continue execution of the calling function, it returns immediately, displaying the error message in the interactive session:

```
function verbose_fussy_sqrt(x)
  println("before fussy_sqrt")
  r = fussy_sqrt(x)
  println("after fussy_sqrt")
  return r
end

julia> verbose_fussy_sqrt(2)
before fussy_sqrt
after fussy_sqrt
1.4142135623730951

julia> verbose_fussy_sqrt(-1)
before fussy_sqrt
negative x not allowed
```

Now suppose we want to handle this circumstance rather than just giving up with an error. To catch an error, you use the `try` and `catch` keywords. Here is a rather contrived example that computes the square root of the absolute value of `x` by handling the error raised by `fussy_sqrt`:

```
function sqrt_abs(x)
  try
    fussy_sqrt(x)
  catch
    fussy_sqrt(-x)
  end
end

julia> sqrt_abs(2)
1.4142135623730951

julia> sqrt_abs(-2)
1.4142135623730951
```

Of course, it would be far simpler and more efficient to just return `sqrt(abs(x))`. However, this demonstrates how `try` and `catch` operate: the `try` block is executed initially, and the value of the entire construct is the value of the last expression if no exceptions are thrown during execution; if an exception is thrown during the evaluation of the `try` block, however, execution of the `try` code ceases immediately and the `catch` block is evaluated instead. If the `catch` block succeeds without incident (it can in turn raise an exception, which would unwind the call stack further), the value of the entire `try-catch` construct is that of the last expression in the `catch` block.

Throw versus Error

The `error` function is convenient for indicating that an error has occurred, but it is built on a more fundamental function: `throw`. Perhaps `throw` should be introduced first, but typical usage calls for `error`, so we have deferred

the introduction of `throw`. Above, we use a form of the `try-catch` expression in which no value is captured by the `catch` block, but there is another form:

```
try
  # execute some code
catch x
  # do something with x
end
```

In this form, if the built-in `throw` function is called by the "execute some code" expression, or any callee thereof, the catch block is executed with the argument of the `throw` function bound to the variable `x`. The `error` function is simply a convenience which always throws an instance of the type `ErrorException`. Here we can see that the object thrown when a divide-by-zero error occurs is of type `DivideByZeroError`:

```
julia> div(1,0)
error: integer divide by zero

julia> try
         div(1,0)
       catch x
         println(typeof(x))
       end
DivideByZeroError
```

`DivideByZeroError` is a concrete subtype of `Exception`, thrown to indicate that an integer division by zero has occurred. Floating-point functions, on the other hand, can simply return `NaN` rather than throwing an exception.

Unlike `error`, which should only be used to indicate an unexpected condition, `throw` is merely a control construct, and can be used to pass any value back to an enclosing `try-catch`:

```
julia> try
         throw("Hello, world.")
       catch x
         println(x)
       end
Hello, world.
```

This example is contrived, of course — the power of the `try-catch` construct lies in the ability to unwind a deeply nested computation immediately to a much higher level in the stack of calling functions. There are situations where no error has occurred, but the ability to unwind the stack and pass a value to a higher level is desirable. These are the circumstances in which `throw` should be used rather than `error`.

1.8.6 Tasks (aka Coroutines)

Tasks are a control flow feature that allows computations to be suspended and resumed in a flexible manner. This feature is sometimes called by other names, such as symmetric coroutines, lightweight threads, cooperative multitasking, or one-shot continuations.

When a piece of computing work (in practice, executing a particular function) is designated as a `Task`, it becomes possible to interrupt it by switching to another `Task`. The original `Task` can later be resumed, at which point it will pick up right where it left off. At first, this may seem similar to a function call. However there are two key differences. First, switching tasks does not use any space, so any number of task switches can occur without consuming the call stack. Second, you may switch among tasks in any order, unlike function calls, where the called function must finish executing before control returns to the calling function.

This kind of control flow can make it much easier to solve certain problems. In some problems, the various pieces of required work are not naturally related by function calls; there is no obvious "caller" or "callee" among the jobs that need to be done. An example is the producer-consumer problem, where one complex procedure is generating values

and another complex procedure is consuming them. The consumer cannot simply call a producer function to get a value, because the producer may have more values to generate and so might not yet be ready to return. With tasks, the producer and consumer can both run as long as they need to, passing values back and forth as necessary.

Julia provides the functions `produce` and `consume` for solving this problem. A producer is a function that calls `produce` on each value it needs to produce:

```
function producer()
  produce("start")
  for n=1:4
    produce(2n)
  end
  produce("stop")
end
```

To consume values, first the producer is wrapped in a `Task`, then `consume` is called repeatedly on that object:

```
julia> p = Task(producer)
Task

julia> consume(p)
"start"

julia> consume(p)
2

julia> consume(p)
4

julia> consume(p)
6

julia> consume(p)
8

julia> consume(p)
"stop"
```

One way to think of this behavior is that `producer` was able to return multiple times. Between calls to `produce`, the producer's execution is suspended and the consumer has control.

A Task can be used as an iterable object in a `for` loop, in which case the loop variable takes on all the produced values:

```
julia> for x in Task(producer)
         println(x)
       end
start
2
4
6
8
stop
```

Note that the `Task()` constructor expects a 0-argument function. A common pattern is for the producer to be parameterized, in which case a partial function application is needed to create a 0-argument *anonymous function*. This can be done either directly or by use of a convenience macro:

```
function mytask(myarg)
    ...
```

```
end

taskHdl = Task(() -> mytask(7))
# or, equivalently
taskHdl = @task mytask(7)
```

`produce` and `consume` are intended for multitasking, and do not launch threads that can run on separate CPUs. True kernel threads are discussed under the topic of *Parallel Computing*.

1.9 Variables and Scoping

Until now, we have simply used variables without any explanation. Julia's usage of variables closely resembles that of other dynamic languages, so we have hopefully gotten away with this liberty. In what follows, however, we address this oversight and provide details of how variables are used, declared, and scoped in Julia.

The *scope* of a variable is the region of code within which a variable is visible. Variable scoping helps avoid variable naming conflicts. The concept is intuitive: two functions can both have arguments called `x` without the two `x`'s referring to the same thing. Similarly there are many other cases where different blocks of code can use the same name without referring to the same thing. The rules for when the same variable name does or doesn't refer to the same thing are called scope rules; this section spells them out in detail.

Certain constructs in the language introduce *scope blocks*, which are regions of code that are eligible to be the scope of some set of variables. The scope of a variable cannot be an arbitrary set of source lines, but will always line up with one of these blocks. The constructs introducing such blocks are:

- `function` bodies (either syntax)
- `while` loops
- `for` loops
- `try` blocks
- `catch` blocks
- `let` blocks
- `type` blocks.

Notably missing from this list are *begin blocks*, which do *not* introduce a new scope block.

Certain constructs introduce new variables into the current innermost scope. When a variable is introduced into a scope, it is also inherited by all inner scopes unless one of those inner scopes explicitly overrides it. These constructs which introduce new variables into the current scope are as follows:

- A declaration `local x` introduces a new local variable.
- A declaration `global x` makes `x` in the current scope and inner scopes refer to the global variable of that name.
- A function's arguments are introduced as new local variables into the function's body scope.
- An assignment `x = y` introduces a new local variable `x` only if `x` is neither declared global nor explicitly introduced as local by any enclosing scope, before or *after* the current line of code.

In the following example, there is only one `x` assigned both inside and outside a loop:

```
function foo(n)
  x = 0
  for i = 1:n
    x = x + 1
```

```
  end
  x
end

julia> foo(10)
10
```

In the next example, the loop has a separate `x` and the function always returns zero:

```
function foo(n)
  x = 0
  for i = 1:n
    local x
    x = i
  end
  x
end

julia> foo(10)
0
```

In this example, an `x` exists only inside the loop, and the function encounters an undefined variable error on its last line (unless there is a global variable `x`):

```
function foo(n)
  for i = 1:n
    x = i
  end
  x
end

julia> foo(10)
in foo: x not defined
```

A variable that is not assigned to or otherwise introduced locally defaults to global, so this function would return the value of the global `x` if there is such a variable, or produce an error if no such global exists. As a consequence, the only way to assign to a global variable inside a non-top-level scope is to explicitly declare the variable as global within some scope, since otherwise the assignment would introduce a new local rather than assigning to the global. This rule works out well in practice, since the vast majority of variables assigned inside functions are intended to be local variables, and using global variables should be the exception rather than the rule, especially assigning new values to them.

One last example shows that an outer assignment introducing `x` need not come before an inner usage:

```
function foo(n)
  f = y -> n + x + y
  x = 1
  f(2)
end

julia> foo(10)
13
```

This last example may seem slightly odd for a normal variable, but allows for named functions — which are just normal variables holding function objects — to be used before they are defined. This allows functions to be defined in whatever order is intuitive and convenient, rather than forcing bottom up ordering or requiring forward declarations, both of which one typically sees in C programs. As an example, here is an inefficient, mutually recursive way to test if positive integers are even or odd:

```
even(n) = n == 0 ? true  :  odd(n-1)
odd(n)  = n == 0 ? false : even(n-1)

julia> even(3)
false

julia> odd(3)
true
```

Julia provides built-in, efficient functions to test this called `iseven` and `isodd` so the above definitions should only be taken as examples.

Since functions can be used before they are defined, as long as they are defined by the time they are actually called, no syntax for forward declarations is necessary, and definitions can be ordered arbitrarily.

At the interactive prompt, variable scope works the same way as anywhere else. The prompt behaves as if there is scope block wrapped around everything you type, except that this scope block is identified with the global scope. This is especially apparent in the case of assignments:

```
julia> for i = 1:1; y = 10; end

julia> y
y not defined

julia> y = 0
0

julia> for i = 1:1; y = 10; end

julia> y
10
```

In the former case, `y` only exists inside of the `for` loop. In the latter case, an outer `y` has been introduced and so is inherited within the loop. Due to the special identification of the prompt's scope block with the global scope, it is not necessary to declare `global y` inside the loop. However, in code not entered into the interactive prompt this declaration would be necessary in order to modify a global variable.

The `let` statement provides a different way to introduce variables. Unlike assignments to local variables, `let` statements allocate new variable bindings each time they run. An assignment modifies an existing value location, and `let` creates new locations. This difference is usually not important, and is only detectable in the case of variables that outlive their scope via closures. The `let` syntax accepts a comma-separated series of assignments and variable names:

```
let var1 = value1, var2, var3 = value3
    code
end
```

Unlike local variable assignments, the assignments do not occur in order. Rather, all assignment right-hand sides are evaluated in the scope outside the `let`, then the `let` variables are assigned "simultaneously". In this way, `let` operates like a function call. Indeed, the following code:

```
let a = b, c = d
  body
end
```

is equivalent to `((a,c)->body)(b, d)`. Therefore it makes sense to write something like `let x = x` since the two `x` variables are distinct and have separate storage. Here is an example where the behavior of `let` is needed:

```
Fs = cell(2);
for i = 1:2
  Fs[i] = ()->i
end

julia> Fs[1]()
2

julia> Fs[2]()
2
```

Here we create and store two closures that return variable `i`. However, it is always the same variable `i`, so the two closures behave identically. We can use `let` to create a new binding for `i`:

```
Fs = cell(2);
for i = 1:2
  let i = i
    Fs[i] = ()->i
  end
end

julia> Fs[1]()
1

julia> Fs[2]()
2
```

Since the `begin` construct does not introduce a new block, it can be useful to use the zero-argument `let` to just introduce a new scope block without creating any new bindings:

```
julia> begin
         local x = 1
         begin
           local x = 2
         end
         x
       end
syntax error: local x declared twice

julia> begin
         local x = 1
         let
           local x = 2
         end
         x
       end
1
```

The first example is illegal because you cannot declare the same variable as local in the same scope twice. The second example is legal since the `let` introduces a new scope block, so the inner local `x` is a different variable than the outer local `x`.

1.9.1 Constants

A common use of variables is giving names to specific, unchanging values. Such variables are only assigned once. This intent can be conveyed to the compiler using the `const` keyword:

```
const e  = 2.71828182845904523536
const pi = 3.14159265358979323846
```

The `const` declaration is allowed on both global and local variables, but is especially useful for globals. It is difficult for the compiler to optimize code involving global variables, since their values (or even their types) might change at almost any time. If a global variable will not change, adding a `const` declaration solves this performance problem.

Local constants are quite different. The compiler is able to determine automatically when a local variable is constant, so local constant declarations are not necessary for performance purposes.

Special top-level assignments, such as those performed by the `function` and `type` keywords, are constant by default.

Note that `const` only affects the variable binding; the variable may be bound to a mutable object (such as an array), and that object may still be modified.

1.10 Types

Type systems have traditionally fallen into two quite different camps: static type systems, where every program expression must have a type computable before the execution of the program, and dynamic type systems, where nothing is known about types until run time, when the actual values manipulated by the program are available. Object orientation allows some flexibility in statically typed languages by letting code be written without the precise types of values being known at compile time. The ability to write code that can operate on different types is called polymorphism. All code in classic dynamically typed languages is polymorphic: only by explicitly checking types, or when objects fail to support operations at run-time, are the types of any values ever restricted.

Julia's type system is dynamic, but gains some of the advantages of static type systems by making it possible to indicate that certain values are of specific types. This can be of great assistance in generating efficient code, but even more significantly, it allows method dispatch on the types of function arguments to be deeply integrated with the language. Method dispatch is explored in detail in *Methods*, but is rooted in the type system presented here.

The default behavior in Julia when types are omitted is to allow values to be of any type. Thus, one can write many useful Julia programs without ever explicitly using types. When additional expressiveness is needed, however, it is easy to gradually introduce explicit type annotations into previously "untyped" code. Doing so will typically increase both the performance and robustness of these systems, and perhaps somewhat counterintuitively, often significantly simplify them.

Describing Julia in the lingo of type systems, it is: dynamic, nominative, parametric and dependent. Generic types can be parameterized by other types and by integers, and the hierarchical relationships between types are explicitly declared, rather than implied by compatible structure. One particularly distinctive feature of Julia's type system is that concrete types may not subtype each other: all concrete types are final and may only have abstract types as their supertypes. While this might at first seem unduly restrictive, it has many beneficial consequences with surprisingly few drawbacks. It turns out that being able to inherit behavior is much more important than being able to inherit structure, and inheriting both causes significant difficulties in traditional object-oriented languages. Other high-level aspects of Julia's type system that should be mentioned up front are:

- There is no division between object and non-object values: all values in Julia are true objects having a type that belongs to a single, fully connected type graph, all nodes of which are equally first-class as types.
- There is no meaningful concept of a "compile-time type": the only type a value has is its actual type when the program is running. This is called a "run-time type" in object-oriented languages where the combination of static compilation with polymorphism makes this distinction significant.
- Only values, not variables, have types — variables are simply names bound to values.
- Both abstract and concrete types can be paramaterized by other types and by integers. Type parameters may be completely omitted when they do not need to be explicitly referenced or restricted.

Julia's type system is designed to be powerful and expressive, yet clear, intuitive and unobtrusive. Many Julia programmers may never feel the need to write code that explicitly uses types. Some kinds of programming, however, become clearer, simpler, faster and more robust with declared types.

A Note On Capitalization. There is no semantic significance to capitalization of names in Julia, unlike, for example, Ruby, where identifiers beginning with an uppercase letter (including type names) are constants. By convention, however, the first letter of each word in a Julia type name begins with a capital letter and underscores are not used to separate words. Variables, on the other hand, are conventionally given lowercase names, with word separation indicated by underscores ("_"). In numerical code it is not uncommon to use single-letter uppercase variable names, especially for matrices. Since types rarely have single-letter names, this does not generally cause confusion, although type parameter placeholders (see below) also typically use single-letter uppercase names like T or S.

1.10.1 Type Declarations

The `::` operator can be used to attach type annotations to expressions and variables in programs. There are two primary reasons to do this:

1. As an assertion to help confirm that your program works the way you expect,
2. To provide extra type information to the compiler, which can then improve performance in many cases

The `::` operator is read as "is an instance of" and can be used anywhere to assert that the value of the expression on the left is an instance of the type on the right. When the type on the right is concrete, the value on the left must have that type as its implementation — recall that all concrete types are final, so no implementation is a subtype of any other. When the type is abstract, it suffices for the value to be implemented by a concrete type that is a subtype of the abstract type. If the type assertion is not true, an exception is thrown, otherwise, the left-hand value is returned:

```
julia> (1+2)::FloatingPoint
type error: typeassert: expected FloatingPoint, got Int64

julia> (1+2)::Int
3
```

This allows a type assertion to be attached to any expression in-place.

When attached to a variable, the `::` operator means something a bit different: it declares the variable to always have the specified type, like a type declaration in a statically-typed language such as C. Every value assigned to the variable will be converted to the declared type using the `convert` function:

```
julia> function foo()
         x::Int8 = 1000
         x
       end

julia> foo()
-24

julia> typeof(ans)
Int8
```

This feature is useful for avoiding performance "gotchas" that could occur if one of the assignments to a variable changed its type unexpectedly.

The "declaration" behavior only occurs in specific contexts:

```
x::Int8         # a variable by itself
local x::Int8   # in a local declaration
x::Int8 = 10    # as the left-hand side of an assignment
```

In value contexts, such as `f(x::Int8)`, the `::` is a type assertion again and not a declaration. Note that these declarations cannot be used in global scope currently, in the REPL, since julia does not yet have constant-type globals.

1.10.2 Abstract Types

Abstract types cannot be instantiated, and serve only as nodes in the type graph, thereby describing sets of related concrete types: those concrete types which are their descendants. We begin with abstract types even though they have no instantiation because they are the backbone of the type system: they form the conceptual hierarchy which makes Julia's type system more than just a collection of object implementations.

Recall that in *Números Inteiros e de Ponto Flutuante*, we introduced a variety of concrete types of numeric values: `Int8`, `Uint8`, `Int16`, `Uint16`, `Int32`, `Uint32`, `Int64`, `Uint64`, `Float32`, and `Float64`. These are all bits types, which we will discuss in the next section. Although they have different representation sizes, `Int8`, `Int16`, `Int32` and `Int64` all have in common that they are signed integer types. Likewise `Uint8`, `Uint16`, `Uint32` and `Uint64` are all unsigned integer types, while `Float32` and `Float64` are distinct in being floating-point types rather than integers. It is common for a piece of code to make sense, for example, only if its arguments are some kind of integer, but not really depend on what particular *kind* of integer, as long as the appropriate low-level implementations of integer operations are used. For example, the greatest common denominator algorithm works for all kinds of integers, but will not work for floating-point numbers. Abstract types allow the construction of a hierarchy of types, providing a context into which concrete types can fit. This allows you, for example, to easily program to any type that is an integer, without restricting an algorithm to a specific type of integer.

Abstract types are declared using the `abstract` keyword. The general syntaxes for declaring an abstract type are:

```
abstract «name»
abstract «name» <: «supertype»
```

The `abstract` keyword introduces a new abstract type, whose name is given by `«name»`. This name can be optionally followed by `<:` and an already-existing type, indicating that the newly declared abstract type is a subtype of this "parent" type.

When no supertype is given, the default supertype is `Any` — a predefined abstract type that all objects are instances of and all types are subtypes of. In type theory, `Any` is commonly called "top" because it is at the apex of the type graph. Julia also has a predefined abstract "bottom" type, at the nadir of the type graph, which is called `None`. It is the exact opposite of `Any`: no object is an instance of `None` and all types are supertypes of `None`.

As a specific example, let's consider a subset of the abstract types that make up Julia's numerical hierarchy:

```
abstract Number
abstract Real      <: Number
abstract FloatingPoint <: Real
abstract Integer   <: Real
abstract Signed    <: Integer
abstract Unsigned <: Integer
```

The `Number` type is a direct child type of `Any`, and `Real` is its child. In turn, `Real` has two children (it has more, but only two are shown here; we'll get to the others later): `Integer` and `FloatingPoint`, separating the world into representations of integers and representations of real numbers. Representations of real numbers include, of course, floating-point types, but also include other types, such as Julia's rationals. Hence, `FloatingPoint` is a proper subtype of `Real`, including only floating-point representations of real numbers. Integers are further subdivided into `Signed` and `Unsigned` varieties.

The `<:` operator in general means "is a subtype of", and, used in declarations like this, declares the right-hand type to be an immediate supertype of the newly declared type. It can also be used in expressions as a subtype operator which returns `true` when its left operand is a subtype of its right operand:

```
julia> Integer <: Number
true

julia> Integer <: FloatingPoint
false
```

Since abstract types have no instantiations and serve as no more than nodes in the type graph, there is not much more to say about them until we introduce parametric abstract types later on in Parametric Types.

1.10.3 Bits Types

A bits type is a concrete type whose data consists of plain old bits. Classic examples of bits types are integers and floating-point values. Unlike most languages, Julia lets you declare your own bits types, rather than providing only a fixed set of built-in bits types. In fact, the standard bits types are all defined in the language itself:

```
bitstype 32 Float32 <: FloatingPoint
bitstype 64 Float64 <: FloatingPoint

bitstype 8  Bool <: Integer
bitstype 32 Char <: Integer

bitstype 8  Int8   <: Signed
bitstype 8  Uint8  <: Unsigned
bitstype 16 Int16  <: Signed
bitstype 16 Uint16 <: Unsigned
bitstype 32 Int32  <: Signed
bitstype 32 Uint32 <: Unsigned
bitstype 64 Int64  <: Signed
bitstype 64 Uint64 <: Unsigned
```

The general syntaxes for declaration of a bitstypes are:

```
bitstype «bits» «name»
bitstype «bits» «name» <: «supertype»
```

The number of bits indicates how much storage the type requires and the name gives the new type a name. A bits type can optionally be declared to be a subtype of some supertype. If a supertype is omitted, then the type defaults to having `Any` as its immediate supertype. The declaration of `Bool` above therefore means that a boolean value takes eight bits to store, and has `Integer` as its immediate supertype. Currently, only sizes that are multiples of 8 bits are supported. Therefore, boolean values, although they really need just a single bit, cannot be declared to be any smaller than eight bits.

The types `Bool`, `Int8` and `Uint8` all have identical representations: they are eight-bit chunks of memory. Since Julia's type system is nominative, however, they are not interchangeable despite having identical structure. Another fundamental difference between them is that they have different supertypes: `Bool`'s direct supertype is `Integer`, `Int8`'s is `Signed`, and `Uint8`'s is `Unsigned`. All other differences between `Bool`, `Int8`, and `Uint8` are matters of behavior — the way functions are defined to act when given objects of these types as arguments. This is why a nominative type system is necessary: if structure determined type, which in turn dictates behavior, then it would be impossible to make `Bool` behave any differently than `Int8` or `Uint8`.

1.10.4 Composite Types

Composite types are called records, structures ("structs" in C), or objects in various languages. A composite type is a collection of named fields, an instance of which can be treated as a single value. In many languages, composite types are the only kind of user-definable type, and they are by far the most commonly used user-defined type in Julia as

well. In mainstream object oriented languages, such as C++, Java, Python and Ruby, composite types also have named functions associated with them, and the combination is called an "object". In purer object-oriented languages, such as Python and Ruby, all values are objects whether they are composites or not. In less pure object oriented languages, including C++ and Java, some values, such as integers and floating-point values, are not objects, while instances of user-defined composite types are true objects with associated methods. In Julia, all values are objects, as in Python and Ruby, but functions are not bundled with the objects they operate on. This is necessary since Julia chooses which method of a function to use by multiple dispatch, meaning that the types of *all* of a function's arguments are considered when selecting a method, rather than just the first one (see *Methods* for more information on methods and dispatch). Thus, it would be inappropriate for functions to "belong" to only their first argument. Organizing methods by association with function objects rather than simply having named bags of methods "inside" each object ends up being a highly beneficial aspect of the language design.

Since composite types are the most common form of user-defined concrete type, they are simply introduced with the `type` keyword followed by a block of field names, optionally annotated with types using the `::` operator:

```
type Foo
  bar
  baz::Int
  qux::Float64
end
```

Fields with no type annotation default to `Any`, and can accordingly hold any type of value.

New objects of composite type `Foo` are created by applying the `Foo` type object like a function to values for its fields:

```
julia> foo = Foo("Hello, world.", 23, 1.5)
Foo("Hello, world.",23,1.5)

julia> typeof(foo)
Foo
```

Since the `bar` field is unconstrained in type, any value will do; the value for `baz` must be an `Int` and `qux` must be a `Float64`. The signature of the default constructor is taken directly from the field type declarations `(Any,Int,Float64)`, so arguments must match this implied type signature:

```
julia> Foo((), 23.5, 1)
no method Foo((),Float64,Int64)
```

You can access the field values of a composite object using the traditional `foo.bar` notation:

```
julia> foo.bar
"Hello, world."

julia> foo.baz
23

julia> foo.qux
1.5
```

You can also change the values as one would expect:

```
julia> foo.qux = 2
2.0

julia> foo.bar = 1//2
1//2
```

Composite types with no fields are singletons; there can be only one instance of such types:

```
type NoFields
end

julia> is(NoFields(), NoFields())
true
```

The `is` function confirms that the "two" constructed instances of `NoFields` are actually one and the same.

There is much more to say about how instances of composite types are created, but that discussion depends on both Parametric Types and on *Methods*, and is sufficiently important to be addressed in its own section: *Constructors*.

1.10.5 Type Unions

A type union is a special abstract type which includes as objects all instances of any of its argument types, constructed using the special `Union` function:

```
julia> IntOrString = Union(Int,String)
Union(Int,String)

julia> 1 :: IntOrString
1

julia> "Hello!" :: IntOrString
"Hello!"

julia> 1.0 :: IntOrString
type error: typeassert: expected Union(Int,String), got Float64
```

The compilers for many languages have an internal union construct for reasoning about types; Julia simply exposes it to the programmer. The union of no types is the "bottom" type, `None`:

```
julia> Union()
None
```

Recall from the discussion above that `None` is the abstract type which is the subtype of all other types, and which no object is an instance of. Since a zero-argument `Union` call has no argument types for objects to be instances of, it should produce a type which no objects are instances of — i.e. `None`.

1.10.6 Tuple Types

Tuples are an abstraction of the arguments of a function — without the function itself. The salient aspects of a function's arguments are their order and their types. The type of a tuple of values is the tuple of types of values:

```
julia> typeof((1,"foo",2.5))
(Int64,ASCIIString,Float64)
```

Accordingly, a tuple of types can be used anywhere a type is expected:

```
julia> (1,"foo",2.5) :: (Int64,String,Any)
(1,"foo",2.5)

julia> (1,"foo",2.5) :: (Int64,String,Float32)
type error: typeassert: expected (Int64,String,Float32), got (Int64,ASCIIString,Float64)
```

If one of the components of the tuple is not a type, however, you will get an error:

```
julia> (1,"foo",2.5) :: (Int64,String,3)
type error: typeassert: expected Type{T}, got (BitsKind,AbstractKind,Int64)
```

Note that the empty tuple `()` is its own type:

```
julia> typeof(())
()
```

1.10.7 Parametric Types

An important and powerful feature of Julia's type system is that it is parametric: types can take parameters, so that type declarations actually introduce a whole family of new types — one for each possible combination of parameter values. There are many languages that support some version of generic programming, wherein data structures and algorithms to manipulate them may be specified without specifying the exact types involved. For example, some form of generic programming exists in ML, Haskell, Ada, Eiffel, C++, Java, C#, F#, and Scala, just to name a few. Some of these languages support true parametric polymorphism (e.g. ML, Haskell, Scala), while others support ad-hoc, template-based styles of generic programming (e.g. C++, Java). With so many different varieties of generic programming and parametric types in various languages, we won't even attempt to compare Julia's parametric types to other languages, but will instead focus on explaining Julia's system in its own right. We will note, however, that because Julia is a dynamically typed language and doesn't need to make all type decisions at compile time, many traditional difficulties encountered in static parametric type systems can be relatively easily handled.

The only kinds of types that are declared are abstract types, bits types, and composite types. All such types can be parameterized, with the same syntax in each case. We will discuss them in in the following order: first, parametric composite types, then parametric abstract types, and finally parametric bits types.

Parametric Composite Types

Type parameters are introduced immediately after the type name, surrounded by curly braces:

```
type Point{T}
  x::T
  y::T
end
```

This declaration defines a new parametric type, `Point{T}`, holding two "coordinates" of type `T`. What, one may ask, is `T`? Well, that's precisely the point of parametric types: it can be any type at all (or an integer, actually, although here it's clearly used as a type). `Point{Float64}` is a concrete type equivalent to the type defined by replacing `T` in the definition of `Point` with `Float64`. Thus, this single declaration actually declares an unlimited number of types: `Point{Float64}`, `Point{String}`, `Point{Int64}`, etc. Each of these is now a usable concrete type:

```
julia> Point{Float64}
Point{Float64}

julia> Point{String}
Point{String}
```

The type `Point{Float64}` is a point whose coordinates are 64-bit floating-point values, while the type `Point{String}` is a "point" whose "coordinates" are string objects (see *Strings*). However, `Point` itself is also a valid type object:

```
julia> Point
Point{T}
```

Here the `T` is the dummy type symbol used in the original declaration of `Point`. What does `Point` by itself mean? It is an abstract type that contains all the specific instances `Point{Float64}`, `Point{String}`, etc.:

```
julia> Point{Float64} <: Point
true

julia> Point{String} <: Point
true
```

Other types, of course, are not subtypes of it:

```
julia> Float64 <: Point
false

julia> String <: Point
false
```

Concrete `Point` types with different values of `T` are never subtypes of each other:

```
julia> Point{Float64} <: Point{Int64}
false

julia> Point{Float64} <: Point{Real}
false
```

This last point is very important:

Even though ‘‘Float64 <: Real‘‘ we DO NOT have ‘‘Point{Float64} <: Point{Real}‘‘.

In other words, in the parlance of type theory, Julia's type parameters are *invariant*, rather than being covariant (or even contravariant). This is for practical reasons: while any instance of `Point{Float64}` may conceptually be like an instance of `Point{Real}` as well, the two types have different representations in memory:

- An instance of `Point{Float64}` can be represented compactly and efficiently as an immediate pair of 64-bit values;
- An instance of `Point{Real}` must be able to hold any pair of instances of `Real`. Since objects that are instances of `Real` can be of arbitrary size and structure, in practice an instance of `Point{Real}` must be represented as a pair of pointers to individually allocated `Real` objects.

The efficiency gained by being able to store `Point{Float64}` objects with immediate values is magnified enormously in the case of arrays: an `Array{Float64}` can be stored as a contiguous memory block of 64-bit floating-point values, whereas an `Array{Real}` must be an array of pointers to individually allocated `Real` objects — which may well be boxed 64-bit floating-point values, but also might be arbitrarily large, complex objects, which are declared to be implementations of the `Real` abstract type.

How does one construct a `Point` object? It is possible to define custom constructors for composite types, which will be discussed in detail in *Constructors*, but in the absence of any special constructor declarations, there are two default ways of creating new composite objects, one in which the type parameters are explicitly given and the other in which they are implied by the arguments to the object constructor.

Since the type `Point{Float64}` is a concrete type equivalent to `Point` declared with `Float64` in place of `T`, it can be applied as a constructor accordingly:

```
julia> Point{Float64}(1.0,2.0)
Point(1.0,2.0)

julia> typeof(ans)
Point{Float64}
```

For the default constructor, exactly one argument must be supplied for each field:

```
julia> Point{Float64}(1.0)
no method Point(Float64,)
```

```
julia> Point{Float64}(1.0,2.0,3.0)
no method Point(Float64,Float64,Float64)
```

The provided arguments need to match the field types exactly, in this case `(Float64,Float64)`, as with all composite type default constructors.

In many cases, it is redundant to provide the type of `Point` object one wants to construct, since the types of arguments to the constructor call already implicitly provide type information. For that reason, you can also apply `Point` itself as a constructor, provided that the implied value of the parameter type `T` is unambiguous:

```
julia> Point(1.0,2.0)
Point(1.0,2.0)

julia> typeof(ans)
Point{Float64}

julia> Point(1,2)
Point(1,2)

julia> typeof(ans)
Point{Int64}
```

In the case of `Point`, the type of `T` is unambiguously implied if and only if the two arguments to `Point` have the same type. When this isn't the case, the constructor will fail with a no method error:

```
julia> Point(1,2.5)
no method Point(Int64,Float64)
```

Constructor methods to appropriately handle such mixed cases can be defined, but that will not be discussed until later on in *Constructors*.

Parametric Abstract Types

Parametric abstract type declarations declare a collection of abstract types, in much the same way:

```
abstract Pointy{T}
```

With this declaration, `Pointy{T}` is a distinct abstract type for each type or integer value of `T`. As with parametric composite types, each such instance is a subtype of `Pointy`:

```
julia> Pointy{Int64} <: Pointy
true

julia> Pointy{1} <: Pointy
true
```

Parametric abstract types are invariant, much as parametric composite types are:

```
julia> Pointy{Float64} <: Pointy{Real}
false

julia> Pointy{Real} <: Pointy{Float64}
false
```

Much as plain old abstract types serve to create a useful hierarchy of types over concrete types, parametric abstract types serve the same purpose with respect to parametric composite types. We could, for example, have declared `Point{T}` to be a subtype of `Pointy{T}` as follows:

```
type Point{T} <: Pointy{T}
  x::T
  y::T
end
```

Given such a declaration, for each choice of `T`, we have `Point{T}` as a subtype of `Pointy{T}`:

```
julia> Point{Float64} <: Pointy{Float64}
true

julia> Point{Real} <: Pointy{Real}
true

julia> Point{String} <: Pointy{String}
true
```

This relationship is also invariant:

```
julia> Point{Float64} <: Pointy{Real}
false
```

What purpose do parametric abstract types like `Pointy` serve? Consider if we create a point-like implementation that only requires a single coordinate because the point is on the diagonal line $x = y$:

```
type DiagPoint{T} <: Pointy{T}
  x::T
end
```

Now both `Point{Float64}` and `DiagPoint{Float64}` are implementations of the `Pointy{Float64}` abstraction, and similarly for every other possible choice of type `T`. This allows programming to a common interface shared by all `Pointy` objects, implemented for both `Point` and `DiagPoint`. This cannot be fully demonstrated, however, until we have introduced methods and dispatch in the next section, *Methods*.

There are situations where it may not make sense for type parameters to range freely over all possible types. In such situations, one can constrain the range of `T` like so:

```
abstract Pointy{T<:Real}
```

With such a declaration, it is acceptable to use any type that is a subtype of `Real` in place of `T`, but not types that are not subtypes of `Real`:

```
julia> Pointy{Float64}
Pointy{Float64}

julia> Pointy{Real}
Pointy{Real}

julia> Pointy{String}
type error: Pointy: in T, expected Real, got AbstractKind

julia> Pointy{1}
type error: Pointy: in T, expected Real, got Int64
```

Type parameters for parametric composite types can be restricted in the same manner:

```
type Point{T<:Real} <: Pointy{T}
  x::T
  y::T
end
```

To give a couple of real-world examples of how all this parametric type machinery can be useful, here is the actual definition of Julia's `Rational` type, representing an exact ratio of integers:

```
type Rational{T<:Integer} <: Real
  num::T
  den::T
end
```

It only makes sense to take ratios of integer values, so the parameter type `T` is restricted to being a subtype of `Integer`, and a ratio of integers represents a value on the real number line, so any `Rational` is an instance of the `Real` abstraction.

Singleton Types

There is a special kind of abstract parametric type that must be mentioned here: singleton types. For each type, `T`, the "singleton type" `Type{T}` is an abstract type whose only instance is the object `T`. Since the definition is a little difficult to parse, let's look at some examples:

```
julia> isa(Float64, Type{Float64})
true

julia> isa(Real, Type{Float64})
false

julia> isa(Real, Type{Real})
true

julia> isa(Float64, Type{Real})
false
```

In other words, `isa(A,Type{B})` is true if and only if `A` and `B` are the same object and that object is a type. Without the parameter, `Type` is simply an abstract type which has all type objects as its instances, including, of course, singleton types:

```
julia> isa(Type{Float64},Type)
true

julia> isa(Float64,Type)
true

julia> isa(Real,Type)
true
```

Any object that is not a type is not an instance of `Type`:

```
julia> isa(1,Type)
false

julia> isa("foo",Type)
false
```

Until we discuss *Parametric Methods* and *conversions*, it is difficult to explain the utility of the singleton type construct, but in short, it allows one to specialize function behavior on specific type *values*, rather just kinds of types, which is all that would be possible in the absence of singleton types. This is useful for writing methods (especially parametric ones) whose behavior depends on a type that is given as an explicit argument rather than implied by the type of one of its arguments.

A few popular languages have singleton types, including Haskell, Scala and Ruby. In general usage, the term "singleton type" refers to a type whose only instance is a single value. This meaning applies to Julia's singleton types, but

with that caveat that only type objects have singleton types, whereas in most languages with singleton types, every object has one.

Parametric Bits Types

Bits types can also be declared parametrically. For example, pointers are represented as boxed bits types which would be declared in Julia like this:

```
# 32-bit system:
bitstype 32 Ptr{T}

# 64-bit system:
bitstype 64 Ptr{T}
```

The slightly odd feature of these declarations as compared to typical parametric composite types, is that the type parameter `T` is not used in the definition of the type itself — it is just an abstract tag, essentially defining an entire family of types with identical structure, differentiated only by their type parameter. Thus, `Ptr{Float64}` and `Ptr{Int64}` are distinct types, even though they have identical representations. And of course, all specific pointer types are subtype of the umbrella `Ptr` type:

```
julia> Ptr{Float64} <: Ptr
true

julia> Ptr{Int64} <: Ptr
true
```

1.10.8 Type Aliases

Sometimes it is convenient to introduce a new name for an already expressible type. For such occasions, Julia provides the `typealias` mechanism. For example, `Uint` is type aliased to either `Uint32` or `Uint64` as is appropriate for the size of pointers on the system:

```
# 32-bit system:
julia> Uint
Uint32

# 64-bit system:
julia> Uint
Uint64
```

This is accomplished via the following code in `base/boot.jl`:

```
if is(Int,Int64)
    typealias Uint Uint64
else
    typealias Uint Uint32
end
```

Of course, this depends on what `Int` is aliased to — but that is pre-defined to be the correct type — either `Int32` or `Int64`.

For parametric types, `typealias` can be convenient for providing a new parametric types name where one of the parameter choices is fixed. Julia's arrays have type `Array{T,n}` where `T` is the element type and `n` is the number of array dimensions. For convenience, writing `Array{Float64}` allows one to specify the element type without specifying the dimension:

```
julia> Array{Float64,1} <: Array{Float64} <: Array
true
```

However, there is no way to equally simply restrict just the dimension but not the element type. Yet, one often needs to ensure an object is a vector or a matrix (imposing restrictions on the number of dimensions). For that reason, the following type aliases are provided:

```
typealias Vector{T} Array{T,1}
typealias Matrix{T} Array{T,2}
```

Writing `Vector{Float64}` is equivalent to writing `Array{Float64,1}`, and the umbrella type `Vector` has as instances all `Array` objects where the second parameter — the number of array dimensions — is 1, regardless of what the element type is. In languages where parametric types must be always specified in full, this is not especially helpful, but in Julia, this allows one to write just `Matrix` for the abstract type including all two-dimensional dense arrays of any element type.

1.10.9 Operations on Types

Since types in Julia are themselves objects, ordinary functions can operate on them. Some functions that are particularly useful for working with or exploring types have already been introduced, such as the `<:` operator, which indicates whether its left hand operand is a subtype of its right hand operand.

The `isa` function tests if an object is of a given type and returns true or false:

```
julia> isa(1,Int)
true

julia> isa(1,FloatingPoint)
false
```

The `typeof` function, already used throughout the manual in examples, returns the type of its argument. Since, as noted above, types are objects, they also have types, and we can ask what their types are. Here we apply `typeof` to an instance of each of the kinds of types discussed above:

```
julia> typeof(Real)
AbstractKind

julia> typeof(Float64)
BitsKind

julia> typeof(Rational)
CompositeKind

julia> typeof(Union(Real,Float64,Rational))
UnionKind

julia> typeof((Real,Float64,Rational,None))
(AbstractKind,BitsKind,CompositeKind,UnionKind)
```

As you can see, the types of types are called, by convention, "kinds":

- Abstract types have type `AbstractKind`
- Bits types have type `BitsKind`
- Composite types have type `CompositeKind`
- Unions have type `UnionKind`
- Tuples of types have a type that is the tuple of their respective kinds.

What if we repeat the process? What is the type of a kind? Kinds, as it happens, are all composite values and thus all have a type of `CompositeKind`:

```
julia> typeof(AbstractKind)
CompositeKind

julia> typeof(BitsKind)
CompositeKind

julia> typeof(CompositeKind)
CompositeKind

julia> typeof(UnionKind)
CompositeKind
```

The reader may note that `CompositeKind` shares with the empty tuple (see above), the distinction of being its own type (i.e. a fixed point of the `typeof` function). This leads any number of tuple types recursively built with `()` and `CompositeKind` as their only atomic values, which are their own type:

```
julia> typeof(())
()

julia> typeof(CompositeKind)
CompositeKind

julia> typeof(((),))
((),)

julia> typeof((CompositeKind,))
(CompositeKind,)

julia> typeof(((),CompositeKind))
((),CompositeKind)
```

All fixed points of the `typeof` function are like this.

Another operation that applies to some kinds of types is `super`. Only abstract types (`AbstractKind`), bits types (`BitsKind`), and composite types (`CompositeKind`) have a supertype, so these are the only kinds of types that the `super` function applies to:

```
julia> super(Float64)
FloatingPoint

julia> super(Number)
Any

julia> super(String)
Any

julia> super(Any)
Any
```

If you apply `super` to other type objects (or non-type objects), a "no method" error is raised:

```
julia> super(Union(Float64,Int64))
no method super(UnionKind,)

julia> super(None)
no method super(UnionKind,)
```

```
julia> super((Float64,Int64))
no method super((BitsKind,BitsKind),)
```

1.11 Methods

Recall from *Funções* that a function is an object that maps a tuple of arguments to a return value, or throws an exception if no appropriate value can be returned. It is very common for the same conceptual function or operation to be implemented quite differently for different types of arguments: adding two integers is very different from adding two floating-point numbers, both of which are distinct from adding an integer to a floating-point number. Despite their implementation differences, these operations all fall under the general concept of "addition". Accordingly, in Julia, these behaviors all belong to a single object: the + function.

To facilitate using many different implementations of the same concept smoothly, functions need not be defined all at once, but can rather be defined piecewise by providing specific behaviors for certain combinations of argument types and counts. A definition of one possible behavior for a function is called a *method*. Thus far, we have presented only examples of functions defined with a single method, applicable to all types of arguments. However, the signatures of method definitions can be annotated to indicate the types of arguments in addition to their number, and more than a single method definition may be provided. When a function is applied to a particular tuple of arguments, the most specific method applicable to those arguments is applied. Thus, the overall behavior of a function is a patchwork of the behaviors of its various method defintions. If the patchwork is well designed, even though the implementations of the methods may be quite different, the outward behavior of the function will appear seamless and consistent.

The choice of which method to execute when a function is applied is called *dispatch*. Julia allows the dispatch process to choose which of a function's methods to call based on the number of arguments given, and on the types of all of the function's arguments. This is different than traditional object-oriented languages, where dispatch occurs based only on the first argument, which often has a special argument syntax, and is sometimes implied rather than explicitly written as an argument.1 Using all of a function's arguments to choose which method should be invoked, rather than just the first, is known as *multiple dispatch*. Multiple dispatch is particularly useful for mathematical code, where it makes little sense to artificially deem the operations to "belong" to one argument more than any of the others: does the addition operation in `x + y` belong to `x` any more than it does to `y`? The implementation of a mathematical operator generally depends on the types of all of its arguments. Even beyond mathematical operations, however, multiple dispatch ends up being a very powerful and convenient paradigm for structuring and organizing programs.

Footnote 1: In C++ or Java, for example, in a method call like obj.meth(arg1,arg2), the object obj "receives" the method call and is implicitly passed to the method via the *this* keyword, rather then as an explicit method argument. When the current *this* object is the receiver of a method call, it can be omitted altogether, writing just meth(arg1,arg2), with this implied as the receiving object.

1.11.1 Defining Methods

Until now, we have, in our examples, defined only functions with a single method having unconstrained argument types. Such functions behave just like they would in traditional dynamically typed languages. Nevertheless, we have used multiple dispatch and methods almost continually without being aware of it: all of Julia's standard functions and operators, like the aforementioned + function, have many methods defining their behavior over various possible combinations of argument type and count.

When defining a function, one can optionally constrain the types of parameters it is applicable to, using the `::` type-assertion operator, introduced in the section on *Composite Types*:

```
f(x::Float64, y::Float64) = 2x + y
```

This function definition applies only to calls where `x` and `y` are both values of type `Float64`:

```
julia> f(2.0, 3.0)
7.0
```

Applying it to any other types of arguments will result in a "no method" error:

```
julia> f(2.0, 3)
no method f(Float64,Int64)

julia> f(float32(2.0), 3.0)
no method f(Float32,Float64)

julia> f(2.0, "3.0")
no method f(Float64,ASCIIString)

julia> f("2.0", "3.0")
no method f(ASCIIString,ASCIIString)
```

As you can see, the arguments must be precisely of type `Float64`. Other numeric types, such as integers or 32-bit floating-point values, are not automatically converted to 64-bit floating-point, nor are strings parsed as numbers. Because `Float64` is a concrete type and concrete types cannot be subclassed in Julia, such a definition can only be applied to arguments that are exactly of type `Float64`. It may often be useful, however, to write more general methods where the declared parameter types are abstract:

```
f(x::Number, y::Number) = 2x - y

julia> f(2.0, 3)
1.0
```

This method definition applies to any pair of arguments that are instances of `Number`. They need not be of the same type, so long as they are each numeric values. The problem of handling disparate numeric types is delegated to the arithmetic operations in the expression `2x - y`.

To define a function with multiple methods, one simply defines the function multiple times, with different numbers and types of arguments. The first method definition for a function creates the function object, and subsequent method definitions add new methods to the existing function object. The most specific method definition matching the number and types of the arguments will be executed when the function is applied. Thus, the two method definitions above, taken together, define the behavior for `f` over all pairs of instances of the abstract type `Number` — but with a different behavior specific to pairs of `Float64` values. If one of the arguments is a 64-bit float but the other one is not, then the `f(Float64,Float64)` method cannot be called and the more general `f(Number,Number)` method must be used:

```
julia> f(2.0, 3.0)
7.0

julia> f(2, 3.0)
1.0

julia> f(2.0, 3)
1.0

julia> f(2, 3)
1
```

The `2x + y` definition is only used in the first case, while the `2x - y` definition is used in the others. No automatic casting or conversion of function arguments is ever performed: all conversion in Julia is non-magical and completely explicit. *Conversion and Promotion*, however, shows how clever application of sufficiently advanced technology can be indistinguishable from magic. [12]

[12] Arthur C. Clarke, *Profiles of the Future* (1961): Clarke's Third Law.

For non-numeric values, and for fewer or more than two arguments, the function `f` remains undefined, and applying it will still result in a "no method" error:

```
julia> f("foo", 3)
no method f(ASCIIString,Int64)

julia> f()
no method f()
```

You can easily see which methods exist for a function by entering the function object itself in an interactive session:

```
julia> f
Methods for generic function f
f(Float64,Float64)
f(Number,Number)
```

This output tells us that `f` is a function object with two methods: one taking two `Float64` arguments and one taking arguments of type `Number`.

In the absence of a type declaration with `::`, the type of a method parameter is `Any` by default, meaning that it is unconstrained since all values in Julia are instances of the abstract type `Any`. Thus, we can define a catch-all method for `f` like so:

```
julia> f(x,y) = println("Whoa there, Nelly.")

julia> f("foo", 1)
Whoa there, Nelly.
```

This catch-all is less specific than any other possible method definition for a pair of parameter values, so it is only be called on pairs of arguments to which no other method definition applies.

Although it seems a simple concept, multiple dispatch on the types of values is perhaps the single most powerful and central feature of the Julia language. Core operations typically have dozens of methods:

```
julia> +
Methods for generic function +
+(Real,Range{T<:Real}) at range.jl:136
+(Real,Range1{T<:Real}) at range.jl:137
+(Ranges{T<:Real},Real) at range.jl:138
+(Ranges{T<:Real},Ranges{T<:Real}) at range.jl:150
+(Bool,) at bool.jl:45
+(Bool,Bool) at bool.jl:48
+(Int64,Int64) at int.jl:224
+(Int128,Int128) at int.jl:226
+(Union(Array{Bool,N},SubArray{Bool,N,A<:Array{T,N},I<:(Union(Int64,Range1{Int64},Range{Int64})...,)
+{T<:Signed}(T<:Signed,T<:Signed) at int.jl:207
+(Uint64,Uint64) at int.jl:225
+(Uint128,Uint128) at int.jl:227
+{T<:Unsigned}(T<:Unsigned,T<:Unsigned) at int.jl:211
+(Float32,Float32) at float.jl:113
+(Float64,Float64) at float.jl:114
+(Complex{T<:Real},Complex{T<:Real}) at complex.jl:207
+(Rational{T<:Integer},Rational{T<:Integer}) at rational.jl:101
+(Bool,Union(Array{Bool,N},SubArray{Bool,N,A<:Array{T,N},I<:(Union(Int64,Range1{Int64},Range{Int64})
+(Union(Array{Bool,N},SubArray{Bool,N,A<:Array{T,N},I<:(Union(Int64,Range1{Int64},Range{Int64})...,)
+(Char,Char) at char.jl:46
+(Char,Int64) at char.jl:47
+(Int64,Char) at char.jl:48
+{T<:Number}(T<:Number,T<:Number) at promotion.jl:68
+(Number,Number) at promotion.jl:40
```

```
+() at operators.jl:30
+(Number,) at operators.jl:36
+(Any,Any,Any) at operators.jl:44
+(Any,Any,Any,Any) at operators.jl:45
+(Any,Any,Any,Any,Any) at operators.jl:46
+(Any,Any,Any,Any...) at operators.jl:48
+{T}(Ptr{T},Integer) at pointer.jl:52
+(Integer,Ptr{T}) at pointer.jl:54
+{T<:Number}(AbstractArray{T<:Number,N},) at abstractarray.jl:232
+{S,T}(Union(Array{S,N},SubArray{S,N,A<:Array{T,N},I<:(Union(Int64,Range1{Int64},Range{Int64})...,)}),
+{T}(Number,Union(Array{T,N},SubArray{T,N,A<:Array{T,N},I<:(Union(Int64,Range1{Int64},Range{Int64}).
+{T}(Union(Array{T,N},SubArray{T,N,A<:Array{T,N},I<:(Union(Int64,Range1{Int64},Range{Int64})...,)}),N
+{S,T<:Real}(Union(Array{S,N},SubArray{S,N,A<:Array{T,N},I<:(Union(Int64,Range1{Int64},Range{Int64})
+{S<:Real,T}(Ranges{S<:Real},Union(Array{T,N},SubArray{T,N,A<:Array{T,N},I<:(Union(Int64,Range1{Int64
+(BitArray{N},BitArray{N}) at bitarray.jl:922
+(BitArray{N},Number) at bitarray.jl:923
+(Number,BitArray{N}) at bitarray.jl:924
+(BitArray{N},AbstractArray{T,N}) at bitarray.jl:986
+(AbstractArray{T,N},BitArray{N}) at bitarray.jl:987
+{Tv,Ti}(SparseMatrixCSC{Tv,Ti},SparseMatrixCSC{Tv,Ti}) at sparse.jl:536
+(SparseMatrixCSC{Tv,Ti<:Integer},Union(Array{T,N},Number)) at sparse.jl:626
+(Union(Array{T,N},Number),SparseMatrixCSC{Tv,Ti<:Integer}) at sparse.jl:627
```

Multiple dispatch together with the flexible parametric type system give Julia its ability to abstractly express high-level algorithms decoupled from implementation details, yet generate efficient, specialized code to handle each case at run time.

1.11.2 Method Ambiguities

It is possible to define a set of function methods such that there is no unique most specific method applicable to some combinations of arguments:

```
julia> g(x::Float64, y) = 2x + y

julia> g(x, y::Float64) = x + 2y
Warning: New definition g(Any,Float64) is ambiguous with g(Float64,Any).
         Make sure g(Float64,Float64) is defined first.

julia> g(2.0, 3)
7.0

julia> g(2, 3.0)
8.0

julia> g(2.0, 3.0)
7.0
```

Here the call `g(2.0, 3.0)` could be handled by either the `g(Float64, Any)` or the `g(Any, Float64)` method, and neither is more specific than the other. In such cases, Julia warns you about this ambiguity, but allows you to proceed, arbitrarily picking a method. You should avoid method ambiguities by specifying an appropriate method for the intersection case:

```
julia> g(x::Float64, y::Float64) = 2x + 2y

julia> g(x::Float64, y) = 2x + y

julia> g(x, y::Float64) = x + 2y
```

```
julia> g(2.0, 3)
7.0

julia> g(2, 3.0)
8.0

julia> g(2.0, 3.0)
10.0
```

To suppress Julia's warning, the disambiguating method must be defined first, since otherwise the ambiguity exists, if transiently, until the more specific method is defined.

1.11.3 Parametric Methods

Method definitions can optionally have type parameters immediately after the method name and before the parameter tuple:

```
same_type{T}(x::T, y::T) = true
same_type(x,y) = false
```

The first method applies whenever both arguments are of the same concrete type, regardless of what type that is, while the second method acts as a catch-all, covering all other cases. Thus, overall, this defines a boolean function that checks whether its two arguments are of the same type:

```
julia> same_type(1, 2)
true

julia> same_type(1, 2.0)
false

julia> same_type(1.0, 2.0)
true

julia> same_type("foo", 2.0)
false

julia> same_type("foo", "bar")
true

julia> same_type(int32(1), int64(2))
false
```

This kind of definition of function behavior by dispatch is quite common — idiomatic, even — in Julia. Method type parameters are not restricted to being used as the types of parameters: they can be used anywhere a value would be in the signature of the function or body of the function. Here's an example where the method type parameter `T` is used as the type parameter to the parametric type `Vector{T}` in the method signature:

```
julia> myappend{T}(v::Vector{T}, x::T) = [v..., x]

julia> myappend([1,2,3],4)
4-element Int64 Array:
1
2
3
4

julia> myappend([1,2,3],2.5)
```

```
no method myappend(Array{Int64,1},Float64)

julia> myappend([1.0,2.0,3.0],4.0)
[1.0,2.0,3.0,4.0]

julia> myappend([1.0,2.0,3.0],4)
no method myappend(Array{Float64,1},Int64)
```

As you can see, the type of the appended element must match the element type of the vector it is appended to, or a "no method" error is raised. In the following example, the method type parameter `T` is used as the return value:

```
julia> mytypeof{T}(x::T) = T

julia> mytypeof(1)
Int64

julia> mytypeof(1.0)
Float64
```

Just as you can put subtype constraints on type parameters in type declarations (see *Parametric Types*), you can also constrain type parameters of methods:

```
same_type_numeric{T<:Number}(x::T, y::T) = true
same_type_numeric(x::Number, y::Number) = false

julia> same_type_numeric(1, 2)
true

julia> same_type_numeric(1, 2.0)
false

julia> same_type_numeric(1.0, 2.0)
true

julia> same_type_numeric("foo", 2.0)
no method same_type_numeric(ASCIIString,Float64)

julia> same_type_numeric("foo", "bar")
no method same_type_numeric(ASCIIString,ASCIIString)

julia> same_type_numeric(int32(1), int64(2))
false
```

The `same_type_numeric` function behaves much like the `same_type` function defined above, but is only defined for pairs of numbers.

1.11.4 Note on Optional and Named Arguments

As mentioned briefly in *Funções*, optional arguments are implemented as syntax for multiple method definitions. For example, this definition:

```
f(a=1,b=2) = a+2b
```

translates to the following three methods:

```
f(a,b) = a+2b
f(a) = f(a,2)
f() = f(1,2)
```

Named arguments behave quite differently from ordinary positional arguments. In particular, they do not participate in method dispatch. Methods are dispatched based only on positional arguments, with named arguments processed after the matching method is identified.

1.12 Constructors

Constructors are functions that create new objects — specifically, instances of *Composite Types*. In Julia, type objects also serve as constructor functions: they create new instances of themselves when applied to an argument tuple as a function. This much was already mentioned briefly when composite types were introduced. For example:

```
type Foo
  bar
  baz
end

julia> foo = Foo(1,2)
Foo(1,2)

julia> foo.bar
1

julia> foo.baz
2
```

For many types, forming new objects by binding their field values together is all that is ever needed to create instances. There are, however, cases where more functionality is required when creating composite objects. Sometimes invariants must be enforced, either by checking arguments or by transforming them. Recursive data structures, especially those that may be self-referential, often cannot be constructed cleanly without first being created in an incomplete state and then altered programmatically to be made whole, as a separate step from object creation. Sometimes, it's just convenient to be able to construct objects with fewer or different types of parameters than they have fields. Julia's system for object construction addresses all of these cases and more.

1.12.1 Outer Constructor Methods

A constructor is just like any other function in Julia in that its overall behavior is defined by the combined behavior of its methods. Accordingly, you can add functionality to a constructor by simply defining new methods. For example, let's say you want to add a constructor method for `Foo` objects that takes only one argument and uses the given value for both the `bar` and `baz` fields. This is simple:

```
Foo(x) = Foo(x,x)

julia> Foo(1)
Foo(1,1)
```

You could also add a zero-argument `Foo` constructor method that supplies default values for both of the `bar` and `baz` fields:

```
Foo() = Foo(0)

julia> Foo()
Foo(0,0)
```

Here the zero-argument constructor method calls the single-argument constructor method, which in turn calls the automatically provided two-argument constructor method. For reasons that will become clear very shortly, additional constructor methods declared as normal methods like this are called *outer* constructor methods. Outer constructor

methods can only ever create a new instance by calling another constructor method, such as the automatically provided default one.

A Note On Nomenclature. While the term "constructor" generally refers to the entire function which constructs objects of a type, it is common to abuse terminology slightly and refer to specific constructor methods as "constructors". In such situations, it is generally clear from context that the term is used to mean "constructor method" rather than "constructor function", especially as it is often used in the sense of singling out a particular method of the constructor from all of the others.

1.12.2 Inner Constructor Methods

While outer constructor methods succeed in addressing the problem of providing additional convenience methods for constructing objects, they fail to address the other two use cases mentioned in the introduction of this chapter: enforcing invariants, and allowing construction of self-referential objects. For these problems, one needs *inner* constructor methods. An inner constructor method is much like an outer constructor method, with two differences:

1. It is declared inside the block of a type declaration, rather than outside of it like normal methods.
2. It has access to a special locally existent function called `new` that creates objects of the block's type.

For example, suppose one wants to declare a type that holds a pair of real numbers, subject to the constraint that the first number is not greater than the second one. One could declare it like this:

```
type OrderedPair
  x::Real
  y::Real

  OrderedPair(x,y) = x > y ? error("out of order") : new(x,y)
end
```

Now `OrderedPair` objects can only be constructed such that `x <= y`:

```
julia> OrderedPair(1,2)
OrderedPair(1,2)

julia> OrderedPair(2,1)
out of order
 in OrderedPair at none:5
```

You can still reach in and directly change the field values to violate this invariant (support for immutable composites is planned but not yet implemented), but messing around with an object's internals uninvited is considered poor form. You (or someone else) can also provide additional outer constructor methods at any later point, but once a type is declared, there is no way to add more inner constructor methods. Since outer constructor methods can only create objects by calling other constructor methods, ultimately, some inner constructor must be called to create an object. This guarantees that all objects of the declared type must come into existence by a call to one of the inner constructor methods provided with the type, thereby giving some degree of real enforcement of a type's invariants, at least for object creation.

If any inner constructor method is defined, no default constructor method is provided: it is presumed that you have supplied yourself with all the inner constructors you need. The default constructor is equivalent to writing your own inner constructor method that takes all of the object's fields as parameters (constrained to be of the correct type, if the corresponding field has a type), and passes them to `new`, returning the resulting object:

```
type Foo
  bar
  baz

  Foo(bar,baz) = new(bar,baz)
end
```

This declaration has the same effect as the earlier definition of the `Foo` type without an explicit inner constructor method. The following two types are equivalent — one with a default constructor, the other with an explicit constructor:

```
type T1
  x::Int64
end

type T2
  x::Int64
  T2(x::Int64) = new(x)
end

julia> T1(1)
T1(1)

julia> T2(1)
T2(1)

julia> T1(1.0)
no method T1(Float64,)
 in method_missing at /Users/stefan/projects/julia/base/base.jl:58

julia> T2(1.0)
no method T2(Float64,)
 in method_missing at /Users/stefan/projects/julia/base/base.jl:58
```

It is considered good form to provide as few inner constructor methods as possible: only those taking all arguments explicitly and enforcing essential error checking and transformation. Additional convenience constructor methods, supplying default values or auxiliary transformations, should be provided as outer constructors that call the inner constructors to do the heavy lifting. This separation is typically quite natural.

1.12.3 Incomplete Initialization

The final problem which has still not been addressed is construction of self-referential objects, or more generally, recursive data structures. Since the fundamental difficulty may not be immediately obvious, let us briefly explain it. Consider the following recursive type declaration:

```
type SelfReferential
  obj::SelfReferential
end
```

This type may appear innocuous enough, until one considers how to construct an instance of it. If `a` is an instance of `SelfReferential`, then a second instance can be created by the call:

```
b = SelfReferential(a)
```

But how does one construct the first instance when no instance exists to provide as a valid value for its `obj` field? The only solution is to allow creating an incompletely initialized instance of `SelfReferential` with an unassigned `obj` field, and using that incomplete instance as a valid value for the `obj` field of another instance, such as, for example, itself.

To allow for the creation of incompletely initialized objects, Julia allows the `new` function to be called with fewer than the number of fields that the type has, returning an object with the unspecified fields uninitialized. The inner constructor method can then use the incomplete object, finishing its initialization before returning it. Here, for example, we take another crack at defining the `SelfReferential` type, with a zero-argument inner constructor returning instances having `obj` fields pointing to themselves:

```
type SelfReferential
  obj::SelfReferential

  SelfReferential() = (x = new(); x.obj = x)
end
```

We can verify that this constructor works and constructs objects that are, in fact, self-referential:

```
x = SelfReferential();

julia> is(x, x)
true

julia> is(x, x.obj)
true

julia> is(x, x.obj.obj)
true
```

Although it is generally a good idea to return a fully initialized object from an inner constructor, incompletely initialized objects can be returned:

```
type Incomplete
  xx

  Incomplete() = new()
end

julia> z = Incomplete();
```

While you are allowed to create objects with uninitialized fields, any access to an uninitialized field is an immediate error:

```
julia> z.xx
access to undefined reference
```

This prevents uninitialized fields from propagating throughout a program or forcing programmers to continually check for uninitialized fields, the way they have to check for `null` values everywhere in Java: if a field is uninitialized and it is used in any way, an error is thrown immediately so no error checking is required. You can also pass incomplete objects to other functions from inner constructors to delegate their completion:

```
type Lazy
  xx

  Lazy(v) = complete_me(new(), v)
end
```

As with incomplete objects returned from constructors, if `complete_me` or any of its callees try to access the `xx` field of the `Lazy` object before it has been initialized, an error will be thrown immediately.

1.12.4 Parametric Constructors

Parametric types add a few wrinkles to the constructor story. Recall from *Parametric Types* that, by default, instances of parametric composite types can be constructed either with explicitly given type parameters or with type parameters implied by the types of the arguments given to the constructor. Here are some examples:

```
type Point{T<:Real}
  x::T
```

```
  y::T
end

## implicit T ##

julia> Point(1,2)
Point(1,2)

julia> Point(1.0,2.5)
Point(1.0,2.5)

julia> Point(1,2.5)
no method Point(Int64,Float64)
 in method_missing at /Users/stefan/projects/julia/base/base.jl:58

## explicit T ##

julia> Point{Int64}(1,2)
Point(1,2)

julia> Point{Int64}(1.0,2.5)
no method Point(Float64,Float64)
 in method_missing at /Users/stefan/projects/julia/base/base.jl:58

julia> Point{Float64}(1.0,2.5)
Point(1.0,2.5)

julia> Point{Float64}(1,2)
no method Point(Int64,Int64)
 in method_missing at /Users/stefan/projects/julia/base/base.jl:58
```

As you can see, for constructor calls with explicit type parameters, the arguments must match that specific type: `Point{Int64}(1,2)` works, but `Point{Int64}(1.0,2.5)` does not. When the type is implied by the arguments to the constructor call, as in `Point(1,2)`, then the types of the arguments must agree — otherwise the `T` cannot be determined — but any pair of real arguments with matching type may be given to the generic `Point` constructor.

What's really going on here is that `Point`, `Point{Float64}` and `Point{Int64}` are all different constructor functions. In fact, `Point{T}` is a distinct constructor function for each type `T`. Without any explicitly provided inner constructors, the declaration of the composite type `Point{T<:Real}` automatically provides an inner constructor, `Point{T}`, for each possible type `T<:Real`, that behaves just like non-parametric default inner constructors do. It also provides a single general outer `Point` constructor that takes pairs of real arguments, which must be of the same type. This automatic provision of constructors is equivalent to the following explicit declaration:

```
type Point{T<:Real}
  x::T
  y::T

  Point(x::T, y::T) = new(x,y)
end

Point{T<:Real}(x::T, y::T) = Point{T}(x,y)
```

Some features of parametric constructor definitions at work here deserve comment. First, inner constructor declarations always define methods of `Point{T}` rather than methods of the general `Point` constructor function. Since `Point` is not a concrete type, it makes no sense for it to even have inner constructor methods at all. Thus, the inner method declaration `Point(x::T, y::T) = new(x,y)` provides an inner constructor method for each value of `T`. It is thus this method declaration that defines the behavior of constructor calls with explicit type para-

meters like `Point{Int64}(1,2)` and `Point{Float64}(1.0,2.0)`. The outer constructor declaration, on the other hand, defines a method for the general `Point` constructor which only applies to pairs of values of the same real type. This declaration makes constructor calls without explicit type parameters, like `Point(1,2)` and `Point(1.0,2.5)`, work. Since the method declaration restricts the arguments to being of the same type, calls like `Point(1,2.5)`, with arguments of different types, result in "no method" errors.

Suppose we wanted to make the constructor call `Point(1,2.5)` work by "promoting" the integer value `1` to the floating-point value `1.0`. The simplest way to achieve this is to define the following additional outer constructor method:

```
Point(x::Int64, y::Float64) = Point(convert(Float64,x),y)
```

This method uses the `convert` function to explicitly convert `x` to `Float64` and then delegates construction to the general constructor for the case where both arguments are `Float64`. With this method definition what was previously a "no method" error now successfully creates a point of type `Point{Float64}`:

```
julia> Point(1,2.5)
Point(1.0,2.5)

julia> typeof(ans)
Point{Float64}
```

However, other similar calls still don't work:

```
julia> Point(1.5,2)
no method Point(Float64,Int64)
```

For a much more general way of making all such calls work sensibly, see *Conversion and Promotion*. At the risk of spoiling the suspense, we can reveal here that the all it takes is the following outer method definition to make all calls to the general `Point` constructor work as one would expect:

```
Point(x::Real, y::Real) = Point(promote(x,y)...)
```

The `promote` function converts all its arguments to a common type — in this case `Float64`. With this method definition, the `Point` constructor promotes its arguments the same way that numeric operators like + do, and works for all kinds of real numbers:

```
julia> Point(1.5,2)
Point(1.5,2.0)

julia> Point(1,1//2)
Point(1//1,1//2)

julia> Point(1.0,1//2)
Point(1.0,0.5)
```

Thus, while the implicit type parameter constructors provided by default in Julia are fairly strict, it is possible to make them behave in a more relaxed but sensible manner quite easily. Moreover, since constructors can leverage all of the power of the type system, methods, and multiple dispatch, defining sophisticated behavior is typically quite simple.

1.12.5 Case Study: Rational

Perhaps the best way to tie all these pieces together is to present a real world example of a parametric composite type and its constructor methods. To that end, here is beginning of rational.jl, which implements Julia's *Rational Numbers*:

```
type Rational{T<:Integer} <: Real
    num::T
    den::T
```

```
    function Rational(num::T, den::T)
        if num == 0 && den == 0
            error("invalid rational: 0//0")
        end
        g = gcd(den, num)
        num = div(num, g)
        den = div(den, g)
        new(num, den)
    end
end
Rational{T<:Integer}(n::T, d::T) = Rational{T}(n,d)
Rational(n::Integer, d::Integer) = Rational(promote(n,d)...)
Rational(n::Integer) = Rational(n,one(n))

//(n::Integer, d::Integer) = Rational(n,d)
//(x::Rational, y::Integer) = x.num // (x.den*y)
//(x::Integer, y::Rational) = (x*y.den) // y.num
//(x::Complex, y::Real) = complex(real(x)//y, imag(x)//y)
//(x::Real, y::Complex) = x*y'//real(y*y')

function //(x::Complex, y::Complex)
    xy = x*y'
    yy = real(y*y')
    complex(real(xy)//yy, imag(xy)//yy)
end
```

The first line — `type Rational{T<:Int} <: Real` — declares that `Rational` takes one type parameter of an integer type, and is itself a real type. The field declarations `num::T` and `den::T` indicate that the data held in a `Rational{T}` object are a pair of integers of type `T`, one representing the rational value's numerator and the other representing its denominator.

Now things get interesting. `Rational` has a single inner constructor method which checks that both of `num` and `den` aren't zero and ensures that every rational is constructed in "lowest terms" with a non-negative denominator. This is accomplished by dividing the given numerator and denominator values by their greatest common divisor, computed using the `gcd` function. Since `gcd` returns the greatest common divisor of its arguments with sign matching the first argument (`den` here), after this division the new value of `den` is guaranteed to be non-negative. Because this is the only inner constructor for `Rational`, we can be certain that `Rational` objects are always constructed in this normalized form.

`Rational` also provides several outer constructor methods for convenience. The first is the "standard" general constructor that infers the type parameter `T` from the type of the numerator and denominator when they have the same type. The second applies when the given numerator and denominator values have different types: it promotes them to a common type and then delegates construction to the outer constructor for arguments of matching type. The third outer constructor turns integer values into rationals by supplying a value of `1` as the denominator.

Following the outer constructor definitions, we have a number of methods for the `//` operator, which provides a syntax for writing rationals. Before these definitions, `//` is a completely undefined operator with only syntax and no meaning. Afterwards, it behaves just as described in *Rational Numbers* — its entire behavior is defined in these few lines. The first and most basic definition just makes `a//b` construct a `Rational` by applying the `Rational` constructor to `a` and `b` when they are integers. When one of the operands of `//` is already a rational number, we construct a new rational for the resulting ratio slightly differently; this behavior is actually identical to division of a rational with an integer. Finally, applying `//` to complex integral values creates an instance of `Complex{Rational}` — a complex number whose real and imaginary parts are rationals:

```
julia> (1 + 2im)//(1 - 2im)
-3//5 + 4//5im
```

```
julia> typeof(ans)
ComplexPair{Rational{Int64}}

julia> ans <: Complex{Rational}
true
```

Thus, although the `//` operator usually returns an instance of `Rational`, if either of its arguments are complex integers, it will return an instance of `Complex{Rational}` instead. The interested reader should consider perusing the rest of rational.jl: it is short, self-contained, and implements an entire basic Julia type in just a little over a hundred lines of code.

1.13 Conversion and Promotion

Julia has a system for promoting arguments of mathematical operators to a common type, which has been mentioned in various other sections, including *Números Inteiros e de Ponto Flutuante*, *Mathematical Operations*, *Types*, and *Methods*. In this section, we explain how this promotion system works, as well as how to extend it to new types and apply it to functions besides built-in mathematical operators. Traditionally, programming languages fall into two camps with respect to promotion of arithmetic arguments:

- **Automatic promotion for built-in arithmetic types and operators.** In most languages, built-in numeric types, when used as operands to arithmetic operators with infix syntax, such as `+`, `-`, `*`, and `/`, are automatically promoted to a common type to produce the expected results. C, Java, Perl, and Python, to name a few, all correctly compute the sum `1 + 1.5` as the floating-point value `2.5`, even though one of the operands to `+` is an integer. These systems are convenient and designed carefully enough that they are generally all-but-invisible to the programmer: hardly anyone consciously thinks of this promotion taking place when writing such an expression, but compilers and interpreters must perform conversion before addition since integers and floating-point values cannot be added as-is. Complex rules for such automatic conversions are thus inevitably part of specifications and implementations for such languages.
- **No automatic promotion.** This camp includes Ada and ML — very "strict" statically typed languages. In these languages, every conversion must be explicitly specified by the programmer. Thus, the example expression `1 + 1.5` would be a compilation error in both Ada and ML. Instead one must write `real(1) + 1.5`, explicitly converting the integer `1` to a floating-point value before performing addition. Explicit conversion everywhere is so inconvenient, however, that even Ada has some degree of automatic conversion: integer literals are promoted to the expected integer type automatically, and floating-point literals are similarly promoted to appropriate floating-point types.

In a sense, Julia falls into the "no automatic promotion" category: mathematical operators are just functions with special syntax, and the arguments of functions are never automatically converted. However, one may observe that applying mathematical operations to a wide variety of mixed argument types is just an extreme case of polymorphic multiple dispatch — something which Julia's dispatch and type systems are particularly well-suited to handle. "Automatic" promotion of mathematical operands simply emerges as a special application: Julia comes with pre-defined catch-all dispatch rules for mathematical operators, invoked when no specific implementation exists for some combination of operand types. These catch-all rules first promote all operands to a common type using user-definable promotion rules, and then invoke a specialized implementation of the operator in question for the resulting values, now of the same type. User-defined types can easily participate in this promotion system by defining methods for conversion to and from other types, and providing a handful of promotion rules defining what types they should promote to when mixed with other types.

1.13.1 Conversion

Conversion of values to various types is performed by the `convert` function. The `convert` function generally takes two arguments: the first is a type object while the second is a value to convert to that type; the returned value is the

value converted to an instance of given type. The simplest way to understand this function is to see it in action:

```
julia> x = 12
12

julia> typeof(x)
Int64

julia> convert(Uint8, x)
12

julia> typeof(ans)
Uint8

julia> convert(FloatingPoint, x)
12.0

julia> typeof(ans)
Float64
```

Conversion isn't always possible, in which case a no method error is thrown indicating that `convert` doesn't know how to perform the requested conversion:

```
julia> convert(FloatingPoint, "foo")
no method convert(Type{FloatingPoint},ASCIIString)
```

Some languages consider parsing strings as numbers or formatting numbers as strings to be conversions (many dynamic languages will even perform conversion for you automatically), however Julia does not: even though some strings can be parsed as numbers, most strings are not valid representations of numbers, and only a very limited subset of them are.

Defining New Conversions

To define a new conversion, simply provide a new method for `convert`. That's really all there is to it. For example, the method to convert a number to a boolean is simply this:

```
convert(::Type{Bool}, x::Number) = (x!=0)
```

The type of the first argument of this method is a *singleton type*, `Type{Bool}`, the only instance of which is `Bool`. Thus, this method is only invoked when the first argument is the type value `Bool`. When invoked, the method determines whether a numeric value is true or false as a boolean, by comparing it to zero:

```
julia> convert(Bool, 1)
true

julia> convert(Bool, 0)
false

julia> convert(Bool, 1im)
true

julia> convert(Bool, 0im)
false
```

The method signatures for conversion methods are often quite a bit more involved than this example, especially for parametric types. The example above is meant to be pedagogical, and is not the actual julia behaviour. This is the actual implementation in julia:

```
convert{T<:Real}(::Type{T}, z::Complex) = (imag(z)==0 ? convert(T,real(z)) :
                                           throw(InexactError()))

julia> convert(Bool, 1im)
InexactError()
 in convert at complex.jl:40
```

Case Study: Rational Conversions

To continue our case study of Julia's `Rational` type, here are the conversions declared in rational.jl, right after the declaration of the type and its constructors:

```
convert{T<:Int}(::Type{Rational{T}}, x::Rational) = Rational(convert(T,x.num),convert(T,x.den))
convert{T<:Int}(::Type{Rational{T}}, x::Int) = Rational(convert(T,x), convert(T,1))

function convert{T<:Int}(::Type{Rational{T}}, x::FloatingPoint, tol::Real)
    if isnan(x); return zero(T)//zero(T); end
    if isinf(x); return sign(x)//zero(T); end
    y = x
    a = d = one(T)
    b = c = zero(T)
    while true
        f = convert(T,round(y)); y -= f
        a, b, c, d = f*a+c, f*b+d, a, b
        if y == 0 || abs(a/b-x) <= tol
            return a//b
        end
        y = 1/y
    end
end
convert{T<:Int}(rt::Type{Rational{T}}, x::FloatingPoint) = convert(rt,x,eps(x))

convert{T<:FloatingPoint}(::Type{T}, x::Rational) = convert(T,x.num)/convert(T,x.den)
convert{T<:Int}(::Type{T}, x::Rational) = div(convert(T,x.num),convert(T,x.den))
```

The initial four convert methods provide conversions to rational types. The first method converts one type of rational to another type of rational by converting the numerator and denominator to the appropriate integer type. The second method does the same conversion for integers by taking the denominator to be 1. The third method implements a standard algorithm for approximating a floating-point number by a ratio of integers to within a given tolerance, and the fourth method applies it, using machine epsilon at the given value as the threshold. In general, one should have `a//b == convert(Rational{Int64}, a/b)`.

The last two convert methods provide conversions from rational types to floating-point and integer types. To convert to floating point, one simply converts both numerator and denominator to that floating point type and then divides. To convert to integer, one can use the `div` operator for truncated integer division (rounded towards zero).

1.13.2 Promotion

Promotion refers to converting values of mixed types to a single common type. Although it is not strictly necessary, it is generally implied that the common type to which the values are converted can faithfully represent all of the original values. In this sense, the term "promotion" is appropriate since the values are converted to a "greater" type — i.e. one which can represent all of the input values in a single common type. It is important, however, not to confuse this with object-oriented (structural) super-typing, or Julia's notion of abstract super-types: promotion has nothing to do with the type hierarchy, and everything to do with converting between alternate representations. For instance, although every `Int32` value can also be represented as a `Float64` value, `Int32` is not a subtype of `Float64`.

Promotion to a common supertype is performed in Julia by the `promote` function, which takes any number of arguments, and returns a tuple of the same number of values, converted to a common type, or throws an exception if promotion is not possible. The most common use case for promotion is to convert numeric arguments to a common type:

```
julia> promote(1, 2.5)
(1.0,2.5)

julia> promote(1, 2.5, 3)
(1.0,2.5,3.0)

julia> promote(2, 3//4)
(2//1,3//4)

julia> promote(1, 2.5, 3, 3//4)
(1.0,2.5,3.0,0.75)

julia> promote(1.5, im)
(1.5 + 0.0im,0.0 + 1.0im)

julia> promote(1 + 2im, 3//4)
(1//1 + 2//1im,3//4 + 0//1im)
```

Integer values are promoted to the largest type of the integer values. Floating-point values are promoted to largest of the floating-point types. Mixtures of integers and floating-point values are promoted to a floating-point type big enough to hold all the values. Integers mixed with rationals are promoted to rationals. Rationals mixed with floats are promoted to floats. Complex values mixed with real values are promoted to the appropriate kind of complex value.

That is really all there is to using promotions. The rest is just a matter of clever application, the most typical "clever" application being the definition of catch-all methods for numeric operations like the arithmetic operators `+`, `-`, `*` and `/`. Here are some of the the catch-all method definitions given in promotion.jl:

```
+(x::Number, y::Number) = +(promote(x,y)...)
-(x::Number, y::Number) = -(promote(x,y)...)
*(x::Number, y::Number) = *(promote(x,y)...)
/(x::Number, y::Number) = /(promote(x,y)...)
```

These method definitions say that in the absence of more specific rules for adding, subtracting, multiplying and dividing pairs of numeric values, promote the values to a common type and then try again. That's all there is to it: nowhere else does one ever need to worry about promotion to a common numeric type for arithmetic operations — it just happens automatically. There are definitions of catch-all promotion methods for a number of other arithmetic and mathematical functions in promotion.jl, but beyond that, there are hardly any calls to `promote` required in the Julia standard library. The most common usages of `promote` occur in outer constructors methods, provided for convenience, to allow constructor calls with mixed types to delegate to an inner type with fields promoted to an appropriate common type. For example, recall that rational.jl provides the following outer constructor method:

```
Rational(n::Integere, d::Integer) = Rational(promote(n,d)...)
```

This allows calls like the following to work:

```
julia> Rational(int8(15),int32(-5))
-3//1

julia> typeof(ans)
Rational{Int64}
```

For most user-defined types, it is better practice to require programmers to supply the expected types to constructor functions explicitly, but sometimes, especially for numeric problems, it can be convenient to do promotion automatically.

Defining Promotion Rules

Although one could, in principle, define methods for the `promote` function directly, this would require many redundant definitions for all possible permutations of argument types. Instead, the behavior of `promote` is defined in terms of an auxiliary function called `promote_rule`, which one can provide methods for. The `promote_rule` function takes a pair of type objects and returns another type object, such that instances of the argument types will be promoted to the returned type. Thus, by defining the rule:

```
promote_rule(::Type{Float64}, ::Type{Float32} ) = Float64
```

one declares that when 64-bit and 32-bit floating-point values are promoted together, they should be promoted to 64-bit floating-point. The promotion type does not need to be one of the argument types, however; the following promotion rules both occur in Julia's standard library:

```
promote_rule(::Type{Uint8}, ::Type{Int8}) = Int
promote_rule(::Type{Char}, ::Type{Uint8}) = Int32
```

As a general rule, Julia promotes integers to *Int* during computation order to avoid overflow. In the latter case, the result type is `Int32` since `Int32` is large enough to contain all possible Unicode code points, and numeric operations on characters always result in plain old integers unless explicitly cast back to characters (see *Characters*). Also note that one does not need to define both `promote_rule(::Type{A}, ::Type{B})` and `promote_rule(::Type{B}, ::Type{A})` — the symmetry is implied by the way `promote_rule` is used in the promotion process.

The `promote_rule` function is used as a building block to define a second function called `promote_type`, which, given any number of type objects, returns the common type to which those values, as arguments to `promote` should be promoted. Thus, if one wants to know, in absence of actual values, what type a collection of values of certain types would promote to, one can use `promote_type`:

```
julia> promote_type(Int8, Uint16)
Int64
```

Internally, `promote_type` is used inside of `promote` to determine what type argument values should be converted to for promotion. It can, however, be useful in its own right. The curious reader can read the code in promotion.jl, which defines the complete promotion mechanism in about 35 lines.

Case Study: Rational Promotions

Finally, we finish off our ongoing case study of Julia's rational number type, which makes relatively sophisticated use of the promotion mechanism with the following promotion rules:

```
promote_rule{T<:Int}(::Type{Rational{T}}, ::Type{T}) = Rational{T}
promote_rule{T<:Int,S<:Int}(::Type{Rational{T}}, ::Type{S}) = Rational{promote_type(T,S)}
promote_rule{T<:Int,S<:Int}(::Type{Rational{T}}, ::Type{Rational{S}}) = Rational{promote_type(T,S)}
promote_rule{T<:Int,S<:FloatingPoint}(::Type{Rational{T}}, ::Type{S}) = promote_type(T,S)
```

The first rule asserts that promotion of a rational number with its own numerator/denominator type, simply promotes to itself. The second rule says that promoting a rational number with any other integer type promotes to a rational type whose numerator/denominator type is the result of promotion of its numerator/denominator type with the other integer type. The third rule applies the same logic to two different types of rational numbers, resulting in a rational of the promotion of their respective numerator/denominator types. The fourth and final rule dictates that promoting a rational with a float results in the same type as promoting the numerator/denominator type with the float.

This small handful of promotion rules, together with the conversion methods discussed above, are sufficient to make rational numbers interoperate completely naturally with all of Julia's other numeric types — integers, floating-point numbers, and complex numbers. By providing appropriate conversion methods and promotion rules in the same manner, any user-defined numeric type can interoperate just as naturally with Julia's predefined numerics.

1.14 Modules

Modules in Julia are separate global variable workspaces. They are delimited syntactically, inside `module Name ... end`. Modules allow you to create top-level definitions without worrying about name conflicts when your code is used together with somebody else's. Within a module, you can control which names from other modules are visible (via importing), and specify which of your names are intended to be public (via exporting).

The following example demonstrates the major features of modules. It is not meant to be run, but is shown for illustrative purposes:

```
module MyModule
using Lib

export MyType, foo

type MyType
    x
end

bar(x) = 2x
foo(a::MyType) = bar(a.x) + 1

import Base.show
show(io, a::MyType) = print(io, "MyType $(a.x)")
end
```

Note that the style is not to indent the body of the module, since that would typically lead to whole files being indented.

This module defines a type `MyType`, and two functions. Function `foo` and type `MyType` are exported, and so will be available for importing into other modules. Function `bar` is private to `MyModule`.

The statement `using Lib` means that a module called `Lib` will be available for resolving names as needed. When a global variable is encountered that has no definition in the current module, the system will search for it in `Lib` and import it if it is found there. This means that all uses of that global within the current module will resolve to the definition of that variable in `Lib`.

Once a variable is imported this way (or, equivalently, with the `import` keyword), a module may not create its own variable with the same name. Imported variables are read-only; assigning to a global variable always affects a variable owned by the current module, or else raises an error.

Method definitions are a bit special: they do not search modules named in `using` statements. The definition `function foo()` creates a new `foo` in the current module, unless `foo` has already been imported from elsewhere. For example, in `MyModule` above we wanted to add a method to the standard `show` function, so we had to write `import Base.show`.

1.14.1 Modules and files

Files and file names are unrelated to modules; modules are associated only with module expressions. One can have multiple files per module, and multiple modules per file:

```
module Foo

include("file1.jl")
include("file2.jl")

end
```

Including the same code in different modules provides mixin-like behavior. One could use this to run the same code with different base definitions, for example testing code by running it with "safe" versions of some operators:

```
module Normal
include("mycode.jl")
end

module Testing
include("safe_operators.jl")
include("mycode.jl")
end
```

1.14.2 Standard modules

There are three important standard modules: Main, Core, and Base.

Main is the top-level module, and Julia starts with Main set as the current module. Variables defined at the prompt go in Main, and `whos()` lists variables in Main.

Core contains all identifiers considered "built in" to the language, i.e. part of the core language and not libraries. Every module implicitly specifies `using Core`, since you can't do anything without those definitions.

Base is the standard library (the contents of base/). All modules implicitly contain `using Base`, since this is needed in the vast majority of cases.

1.14.3 Default top-level definitions and bare modules

In addition to `using Base`, a module automatically contains a definition of the `eval` function, which evaluates expressions within the context of that module.

If these definitions are not wanted, modules can be defined using the keyword `baremodule` instead. In terms of `baremodule`, a standard `module` looks like this:

```
baremodule Mod
using Base
eval(x) = Core.eval(Mod, x)
eval(m,x) = Core.eval(m, x)
...
end
```

1.14.4 Miscellaneous details

If a name is qualified (e.g. `Base.sin`), then it can be accessed even if it is not exported. This is often useful when debugging.

Macros must be exported if they are intended to be used outside their defining module. Macro names are written with `@` in import and export statements, e.g. `import Mod.@mac`.

The syntax `M.x = y` does not work to assign a global in another module; global assignment is always module-local.

A variable can be "reserved" for the current module without assigning to it by declaring it as `global x` at the top level. This can be used to prevent name conflicts for globals initialized after load time.

1.15 Metaprogramming

The strongest legacy of Lisp in the Julia language is its metaprogramming support. Like Lisp, Julia is homoiconic: it represents its own code as a data structure of the language itself. Since code is represented by objects that can be created and manipulated from within the language, it is possible for a program to transform and generate its own code. This allows sophisticated code generation without extra build steps, and also allows true Lisp-style macros, as compared to preprocessor "macro" systems, like that of C and C++, that perform superficial textual manipulation as a separate pass before any real parsing or interpretation occurs. Another aspect of metaprogramming is reflection: the ability of a running program to dynamically discover properties of itself. Reflection emerges naturally from the fact that all data types and code are represented by normal Julia data structures, so the structure of the program and its types can be explored programmatically just like any other data.

1.15.1 Expressions and Eval

Julia code is represented as a syntax tree built out of Julia data structures of type `Expr`. This makes it easy to construct and manipulate Julia code from within Julia, without generating or parsing source text. Here is the definition of the `Expr` type:

```
type Expr
  head::Symbol
  args::Array{Any,1}
  typ
end
```

The `head` is a symbol identifying the kind of expression, and `args` is an array of subexpressions, which may be symbols referencing the values of variables at evaluation time, may be nested `Expr` objects, or may be actual values of objects. The `typ` field is used by type inference to store type annotations, and can generally be ignored.

There is special syntax for "quoting" code (analogous to quoting strings) that makes it easy to create expression objects without explicitly constructing `Expr` objects. There are two forms: a short form for inline expressions using `:` followed by a single expression, and a long form for blocks of code, enclosed in `quote ... end`. Here is an example of the short form used to quote an arithmetic expression:

```
julia> ex = :(a+b*c+1)
+(a,*(b,c),1)

julia> typeof(ex)
Expr

julia> ex.head
call

julia> typeof(ans)
Symbol

julia> ex.args
4-element Any Array:
  +
  a
  :(*(b,c))
 1

julia> typeof(ex.args[1])
Symbol

julia> typeof(ex.args[2])
```

```
Symbol

julia> typeof(ex.args[3])
Expr

julia> typeof(ex.args[4])
Int64
```

Expressions provided by the parser generally only have symbols, other expressions, and literal values as their args, whereas expressions constructed by Julia code can easily have arbitrary run-time values without literal forms as args. In this specific example, `+` and `a` are symbols, `*(b,c)` is a subexpression, and `1` is a literal 64-bit signed integer. Here's an example of the longer expression quoting form:

```
julia> quote
     x = 1
     y = 2
     x + y
   end

begin
  x = 1
  y = 2
  +(x,y)
end
```

When the argument to `:` is just a symbol, a `Symbol` object results instead of an `Expr`:

```
julia> :foo
foo

julia> typeof(ans)
Symbol
```

In the context of an expression, symbols are used to indicate access to variables, and when an expression is evaluated, a symbol evaluates to the value bound to that symbol in the appropriate scope (see *Variables and Scoping* for further details).

Eval and Interpolation

Given an expression object, one can cause Julia to evaluate (execute) it at the *top level* scope — i.e. in effect like loading from a file or typing at the interactive prompt — using the `eval` function:

```
julia> :(1 + 2)
+(1,2)

julia> eval(ans)
3

julia> ex = :(a + b)
+(a,b)

julia> eval(ex)
a not defined

julia> a = 1; b = 2;

julia> eval(ex)
3
```

Expressions passed to `eval` are not limited to returning values — they can also have side-effects that alter the state of the top-level evaluation environment:

```
julia> ex = :(x = 1)
x = 1

julia> x
x not defined

julia> eval(ex)
1

julia> x
1
```

Here, the evaluation of an expression object causes a value to be assigned to the top-level variable `x`.

Since expressions are just `Expr` objects which can be constructed programmatically and then evaluated, one can, from within Julia code, dynamically generate arbitrary code which can then be run using `eval`. Here is a simple example:

```
julia> a = 1;

julia> ex = Expr(:call, {:+,a,:b}, Any)
:(+(1,b))

julia> a = 0; b = 2;

julia> eval(ex)
3
```

The value of `a` is used to construct the expression `ex` which applies the `+` function to the value 1 and the variable `b`. Note the important distinction between the way `a` and `b` are used:

- The value of the *variable* `a` at expression construction time is used as an immediate value in the expression. Thus, the value of `a` when the expression is evaluated no longer matters: the value in the expression is already `1`, independent of whatever the value of `a` might be.
- On the other hand, the *symbol* `:b` is used in the expression construction, so the value of the variable `b` at that time is irrelevant — `:b` is just a symbol and the variable `b` need not even be defined. At expression evaluation time, however, the value of the symbol `:b` is resolved by looking up the value of the variable `b`.

Constructing `Expr` objects like this is powerful, but somewhat tedious and ugly. Since the Julia parser is already excellent at producing expression objects, Julia allows "splicing" or interpolation of expression objects, prefixed with `$`, into quoted expressions, written using normal syntax. The above example can be written more clearly and concisely using interpolation:

```
julia> a = 1;
1

julia> ex = :($a + b)
:(+(1,b))
```

This syntax is automatically rewritten to the form above where we explicitly called `Expr`. The use of `$` for expression interpolation is intentionally reminiscent of *string interpolation* and *command interpolation*. Expression interpolation allows convenient, readable programmatic construction of complex Julia expressions.

Code Generation

When a significant amount of repetitive boilerplate code is required, it is common to generate it programmatically to avoid redundancy. In most languages, this requires an extra build step, and a separate program to generate the repetitive code. In Julia, expression interpolation and eval allow such code generation to take place in the normal course of program execution. For example, the following code defines a series of operators on three arguments in terms of their 2-argument forms:

```
for op = (:+, :*, :&, :|, :$)
  eval(quote
    ($op)(a,b,c) = ($op)(($op)(a,b),c)
  end)
end
```

In this manner, Julia acts as its own preprocessor, and allows code generation from inside the language. The above code could be written slightly more tersely using the `:` prefix quoting form:

```
for op = (:+, :*, :&, :|, :$)
  eval(:(($op)(a,b,c) = ($op)(($op)(a,b),c)))
end
```

This sort of in-language code generation, however, using the `eval(quote(...))` pattern, is common enough that Julia comes with a macro to abbreviate this pattern:

```
for op = (:+, :*, :&, :|, :$)
  @eval ($op)(a,b,c) = ($op)(($op)(a,b),c)
end
```

The `@eval` macro rewrites this call to be precisely equivalent to the above longer versions. For longer blocks of generated code, the expression argument given to `@eval` can be a block:

```
@eval begin
  # multiple lines
end
```

Interpolating into an unquoted expression is not supported and will cause a compile-time error:

```
julia> $a + b
unsupported or misplaced expression $
```

1.15.2 Macros

Macros are the analogue of functions for expression generation at compile time: they allow the programmer to automatically generate expressions by transforming zero or more argument expressions into a single result expression, which then takes the place of the macro call in the final syntax tree. Macros are invoked with the following general syntax:

```
@name expr1 expr2 ...
@name(expr1, expr2, ...)
```

Note the distinguishing `@` before the macro name and the lack of commas between the argument expressions in the first form, and the lack of whitespace after `@name` in the second form. The two styles should not be mixed. For example, the following syntax is different from the examples above; it passes the tuple `(expr1, expr2, ...)` as one argument to the macro:

```
@name (expr1, expr2, ...)
```

Before the program runs, this statement will be replaced with the result of calling an expander function for `name` on the expression arguments. Expanders are defined with the `macro` keyword:

```
macro name(expr1, expr2, ...)
    ...
end
```

Here, for example, is the definition of Julia's `@assert` macro (see error.jl):

```
macro assert(ex)
    :($ex ? nothing : error("Assertion failed: ", $(string(ex))))
end
```

This macro can be used like this:

```
julia> @assert 1==1.0

julia> @assert 1==0
Assertion failed: 1==0
```

Macro calls are expanded so that the above calls are precisely equivalent to writing:

```
1==1.0 ? nothing : error("Assertion failed: ", "1==1.0")
1==0 ? nothing : error("Assertion failed: ", "1==0")
```

That is, in the first call, the expression `:(1==1.0)` is spliced into the test condition slot, while the value of `string(:(1==1.0))` is spliced into the assertion message slot. The entire expression, thus constructed, is placed into the syntax tree where the `@assert` macro call occurs. Therefore, if the test expression is true when evaluated, the entire expression evaluates to nothing, whereas if the test expression is false, an error is raised indicating the asserted expression that was false. Notice that it would not be possible to write this as a function, since only the *value* of the condition and not the expression that computed it would be available.

The `@assert` example also shows how macros can include a `quote` block, which allows for convenient manipulation of expressions inside the macro body.

Hygiene

An issue that arises in more complex macros is that of hygiene. In short, Julia must ensure that variables introduced and used by macros do not accidentally clash with the variables used in code interpolated into those macros. Another concern arises from the fact that a macro may be called in a different module from where it was defined. In this case we need to ensure that all global variables are resolved to the correct module.

To demonstrate these issues, let us consider writing a `@time` macro that takes an expression as its argument, records the time, evaluates the expression, records the time again, prints the difference between the before and after times, and then has the value of the expression as its final value. The macro might look like this:

```
macro time(ex)
  quote
    local t0 = time()
    local val = $ex
    local t1 = time()
    println("elapsed time: ", t1-t0, " seconds")
    val
  end
end
```

Here, we want `t0`, `t1`, and `val` to be private temporary variables, and we want `time` to refer to the `time` function in the standard library, not to any `time` variable the user might have (the same applies to `println`). Imagine the

problems that could occur if the user expression `ex` also contained assignments to a variable called `t0`, or defined its own `time` variable. We might get errors, or mysteriously incorrect behavior.

Julia's macro expander solves these problems in the following way. First, variables within a macro result are classified as either local or global. A variable is considered local if it is assigned to (and not declared global), declared local, or used as a function argument name. Otherwise, it is considered global. Local variables are then renamed to be unique (using the `gensym` function, which generates new symbols), and global variables are resolved within the macro definition environment. Therefore both of the above concerns are handled; the macro's locals will not conflict with any user variables, and `time` and `println` will refer to the standard library definitions.

One problem remains however. Consider the following use of this macro:

```
module MyModule
import Base.@time

time() = ... # compute something

@time time()
end
```

Here the user expression `ex` is a call to `time`, but not the same `time` function that the macro uses. It clearly refers to `MyModule.time`. Therefore we must arrange for the code in `ex` to be resolved in the macro call environment. This is done by "escaping" the expression with the `esc` function:

```
macro time(ex)
    ...
    local val = $(esc(ex))
    ...
end
```

An expression wrapped in this manner is left alone by the macro expander and simply pasted into the output verbatim. Therefore it will be resolved in the macro call environment.

This escaping mechanism can be used to "violate" hygiene when necessary, in order to introduce or manipulate user variables. For example, the following macro sets `x` to zero in the call environment:

```
macro zerox()
  esc(:(x = 0))
end

function foo()
  x = 1
  @zerox
  x  # is zero
end
```

This kind of manipulation of variables should be used judiciously, but is occasionally quite handy.

Non-Standard String Literals

Recall from *Strings* that string literals prefixed by an identifier are called non-standard string literals, and can have different semantics than un-prefixed string literals. For example:

- `E"$100\n"` interprets escape sequences but does no string interpolation
- `r"^\s*(?:#|$)"` produces a regular expression object rather than a string
- `b"DATA\xff\u2200"` is a byte array literal for `[68,65,84,65,255,226,136,128]`.

Perhaps surprisingly, these behaviors are not hard-coded into the Julia parser or compiler. Instead, they are custom behaviors provided by a general mechanism that anyone can use: prefixed string literals are parsed as calls to specially-named macros. For example, the regular expression macros is just the following:

```
macro r_str(p)
  Regex(p)
end
```

That's all. This macro says that the literal contents of the string literal `r"^\s*(?:#|$)"` should be passed to the `@r_str` macro and the result of that expansion should be placed in the syntax tree where the string literal occurs. In other words, the expression `r"^\s*(?:#|$)"` is equivalent to placing the following object directly into the syntax tree:

```
Regex("^\\s*(?:#|\$)")
```

Not only is the string literal form shorter and far more convenient, but it is also more efficient: since the regular expression is compiled and the `Regex` object is actually created *when the code is compiled*, the compilation occurs only once, rather than every time the code is executed. Consider if the regular expression occurs in a loop:

```
for line = lines
  m = match(r"^\s*(?:#|$)", line)
  if m.match == nothing
    # non-comment
  else
    # comment
  end
end
```

Since the regular expression `r"^\s*(?:#|$)"` is compiled and inserted into the syntax tree when this code is parsed, the expression is only compiled once instead of each time the loop is executed. In order to accomplish this without macros, one would have to write this loop like this:

```
re = Regex("^\\s*(?:#|\$)")
for line = lines
  m = match(re, line)
  if m.match == nothing
    # non-comment
  else
    # comment
  end
end
```

Moreover, if the compiler could not determine that the regex object was constant over all loops, certain optimizations might not be possible, making this version still less efficient than the more convenient literal form above. Of course, there are still situations where the non-literal form is more convenient: if one needs to interpolate a variable into the regular expression, has to take this more verbose approach; in cases where the regular expression pattern itself is dynamic, potentially changing upon each loop iteration, a new regular expression object must be constructed on each iteration. The vast majority of use cases, however, one does not construct regular expressions dynamically, depending on run-time data. In this majority of cases, the ability to write regular expressions as compile-time values is, well, invaluable.

The mechanism for user-defined string literals is deeply, profoundly powerful. Not only are Julia's non-standard literals implemented using it, but also the command literal syntax (`` `echo "Hello, $person"` ``) is implemented with the following innocuous-looking macro:

```
macro cmd(str)
  :(cmd_gen($shell_parse(str)))
end
```

Of course, a large amount of complexity is hidden in the functions used in this macro definition, but they are just functions, written entirely in Julia. You can read their source and see precisely what they do — and all they do is construct expression objects to be inserted into your program's syntax tree.

1.15.3 Reflection

1.16 Arrays

Julia, like most technical computing languages, provides a first-class array implementation. Most technical computing languages pay a lot of attention to their array implementation at the expense of other containers. Julia does not treat arrays in any special way. The array library is implemented almost completely in Julia itself, and derives its performance from the compiler, just like any other code written in Julia.

An array is a collection of objects stored in a multi-dimensional grid. In the most general case, an array may contain objects of type `Any`. For most computational purposes, arrays should contain objects of a more specific type, such as `Float64` or `Int32`.

In general, unlike many other technical computing languages, Julia does not expect programs to be written in a vectorized style for performance. Julia's compiler uses type inference and generates optimized code for scalar array indexing, allowing programs to be written in a style that is convenient and readable, without sacrificing performance, and using less memory at times.

In Julia, all arguments to functions are passed by reference. Some technical computing languages pass arrays by value, and this is convenient in many cases. In Julia, modifications made to input arrays within a function will be visible in the parent function. The entire Julia array library ensures that inputs are not modified by library functions. User code, if it needs to exhibit similar behaviour, should take care to create a copy of inputs that it may modify.

1.16.1 Basic Functions

1. `ndims(A)` — the number of dimensions of A
2. `size(A,n)` — the size of A in a particular dimension
3. `size(A)` — a tuple containing the dimensions of A
4. `eltype(A)` — the type of the elements contained in A
5. `length(A)` — the number of elements in A
6. `nnz(A)` — the number of nonzero values in A
7. `stride(A,k)` — the size of the stride along dimension k
8. `strides(A)` — a tuple of the linear index distances between adjacent elements in each dimension

1.16.2 Construction and Initialization

Many functions for constructing and initializing arrays are provided. In the following list of such functions, calls with a `dims...` argument can either take a single tuple of dimension sizes or a series of dimension sizes passed as a variable number of arguments.

1. `Array(type, dims...)` — an uninitialized dense array
2. `cell(dims...)` — an uninitialized cell array (heterogeneous array)
3. `zeros(type, dims...)` — an array of all zeros of specified type

4. `ones(type, dims...)` — an array of all ones of specified type
5. `trues(dims...)` — a `Bool` array with all values `true`
6. `falses(dims...)` — a `Bool` array with all values `false`
7. `reshape(A, dims...)` — an array with the same data as the given array, but with different dimensions.
8. `copy(A)` — copy `A`
9. `deepcopy(A)` — copy `A`, recursively copying its elements
10. `similar(A, element_type, dims...)` — an uninitialized array of the same type as the given array (dense, sparse, etc.), but with the specified element type and dimensions. The second and third arguments are both optional, defaulting to the element type and dimensions of `A` if omitted.
11. `reinterpret(type, A)` — an array with the same binary data as the given array, but with the specified element type.
12. `rand(dims)` — random array with `Float64` uniformly distributed values in [0,1)
13. `randf(dims)` — random array with `Float32` uniformly distributed values in [0,1)
14. `randn(dims)` — random array with `Float64` normally distributed random values with a mean of 0 and standard deviation of 1
15. `eye(n)` — n-by-n identity matrix
16. `eye(m, n)` — m-by-n identity matrix
17. `linspace(start, stop, n)` — a vector of `n` linearly-spaced elements from `start` to `stop`.
18. `fill!(A, x)` — fill the array `A` with value `x`

The last function, `fill!`, is different in that it modifies an existing array instead of constructing a new one. As a convention, functions with this property have names ending with an exclamation point. These functions are sometimes called "mutating" functions, or "in-place" functions.

1.16.3 Comprehensions

Comprehensions provide a general and powerful way to construct arrays. Comprehension syntax is similar to set construction notation in mathematics:

```
A = [ F(x,y,...) for x=rx, y=ry, ... ]
```

The meaning of this form is that `F(x,y,...)` is evaluated with the variables `x`, `y`, etc. taking on each value in their given list of values. Values can be specified as any iterable object, but will commonly be ranges like `1:n` or `2:(n-1)`, or explicit arrays of values like `[1.2, 3.4, 5.7]`. The result is an N-d dense array with dimensions that are the concatenation of the dimensions of the variable ranges `rx`, `ry`, etc. and each `F(x,y,...)` evaluation returns a scalar.

The following example computes a weighted average of the current element and its left and right neighbour along a 1-d grid.

```
julia> const x = rand(8)
8-element Float64 Array:
 0.276455
 0.614847
 0.0601373
 0.896024
 0.646236
 0.143959
 0.0462343
```

```
 0.730987

julia> [ 0.25*x[i-1] + 0.5*x[i] + 0.25*x[i+1] for i=2:length(x)-1 ]
6-element Float64 Array:
 0.391572
 0.407786
 0.624605
 0.583114
 0.245097
 0.241854
```

NOTE: In the above example, `x` is declared as constant because type inference in Julia does not work as well on non-constant global variables.

The resulting array type is inferred from the expression; in order to control the type explicitly, the type can be prepended to the comprehension. For example, in the above example we could have avoided declaring `x` as constant, and ensured that the result is of type `Float64` by writing:

```
Float64[ 0.25*x[i-1] + 0.5*x[i] + 0.25*x[i+1] for i=2:length(x)-1 ]
```

Using curly brackets instead of square brackets is a shortand notation for an array of type `Any`:

```
julia> { i/2 for i = 1:3 }
3-element Any Array:
 0.5
 1.0
 1.5
```

1.16.4 Indexing

The general syntax for indexing into an n-dimensional array A is:

```
X = A[I_1, I_2, ..., I_n]
```

where each I_k may be:

1. A scalar value
2. A `Range` of the form `:`, `a:b`, or `a:b:c`
3. An arbitrary integer vector, including the empty vector `[]`
4. A boolean vector

The result X generally has dimensions `(length(I_1), length(I_2), ..., length(I_n))`, with location `(i_1, i_2, ..., i_n)` of X containing the value `A[I_1[i_1], I_2[i_2], ..., I_n[i_n]]`. Trailing dimensions indexed with scalars are dropped. For example, the dimensions of `A[I, 1]` will be `(length(I),)`. The size of a dimension indexed by a boolean vector will be the number of true values in the vector (they behave as if they were transformed with `find`).

Indexing syntax is equivalent to a call to `getindex`:

```
X = getindex(A, I_1, I_2, ..., I_n)
```

Example:

```
julia> x = reshape(1:16, 4, 4)
4x4 Int64 Array
1 5 9 13
2 6 10 14
```

```
3 7 11 15
4 8 12 16

julia> x[2:3, 2:end-1]
2x2 Int64 Array
6 10
7 11
```

1.16.5 Assignment

The general syntax for assigning values in an n-dimensional array A is:

```
A[I_1, I_2, ..., I_n] = X
```

where each I_k may be:

1. A scalar value
2. A `Range` of the form `:`, `a:b`, or `a:b:c`
3. An arbitrary integer vector, including the empty vector `[]`
4. A boolean vector

The size of X should be `(length(I_1), length(I_2), ..., length(I_n))`, and the value in location `(i_1, i_2, ..., i_n)` of A is overwritten with the value `X[I_1[i_1], I_2[i_2], ..., I_n[i_n]]`.

Index assignment syntax is equivalent to a call to `setindex!`:

```
A = setindex!(A, X, I_1, I_2, ..., I_n)
```

Example:

```
julia> x = reshape(1:9, 3, 3)
3x3 Int64 Array
1 4 7
2 5 8
3 6 9

julia> x[1:2, 2:3] = -1
3x3 Int64 Array
1 -1 -1
2 -1 -1
3 6 9
```

1.16.6 Concatenation

Arrays can be concatenated along any dimension using the following syntax:

1. `cat(dim, A...)` — concatenate input n-d arrays along the dimension `dim`
2. `vcat(A...)` — Shorthand for `cat(1, A...)`
3. `hcat(A...)` — Shorthand for `cat(2, A...)`
4. `hvcat(A...)`

Concatenation operators may also be used for concatenating arrays:

1. `[A B C ...]` — calls `hcat`
2. `[A, B, C, ...]` — calls `vcat`
3. `[A B; C D; ...]` — calls `hvcat`

1.16.7 Vectorized Operators and Functions

The following operators are supported for arrays. In case of binary operators, the dot version of the operator should be used when both inputs are non-scalar, and any version of the operator may be used if one of the inputs is a scalar.

1. Unary Arithmetic — -
2. Binary Arithmetic — +, -, *, .*, /, ./, \, .\, ^, .^, div, mod
3. Comparison — ==, !=, <, <=, >, >=
4. Unary Boolean or Bitwise — ~
5. Binary Boolean or Bitwise — &, |, $
6. Trigonometrical functions — sin, cos, tan, sinh, cosh, tanh, asin, acos, atan, atan2, sec, csc, cot, asec, acsc, acot, sech, csch, coth, asech, acsch, acoth, sinc, cosc, hypot
7. Logarithmic functions — log, log2, log10, log1p
8. Exponential functions — exp, expm1, exp2, ldexp
9. Rounding functions — ceil, floor, trunc, round, ipart, fpart
10. Other mathematical functions — min, max, abs, pow, sqrt, cbrt, erf, erfc, gamma, lgamma, real, conj, clamp

1.16.8 Broadcasting

It is sometimes useful to perform element-by-element binary operations on arrays of different sizes, such as adding a vector to each column of a matrix. An inefficient way to do this would be to replicate the vector to the size of the matrix:

```
julia> a = rand(2,1); A = rand(2,3);

julia> repmat(a,1,3)+A
2x3 Float64 Array:
 0.848333  1.66714  1.3262
 1.26743   1.77988  1.13859
```

This is wasteful when dimensions get large, so Julia offers the MATLAB-inspired `bsxfun`, which expands singleton dimensions in array arguments to match the corresponding dimension in the other array without using extra memory, and applies the given binary function:

```
julia> bsxfun(+, a, A)
2x3 Float64 Array:
 0.848333  1.66714  1.3262
 1.26743   1.77988  1.13859

julia> b = rand(1,2)
1x2 Float64 Array:
 0.629799  0.754948

julia> bsxfun(+, a, b)
```

```
2x2 Float64 Array:
 1.31849  1.44364
 1.56107  1.68622
```

1.16.9 Implementation

The base array type in Julia is the abstract type `AbstractArray{T,n}`. It is parametrized by the number of dimensions `n` and the element type `T`. `AbstractVector` and `AbstractMatrix` are aliases for the 1-d and 2-d cases. Operations on `AbstractArray` objects are defined using higher level operators and functions, in a way that is independent of the underlying storage class. These operations are guaranteed to work correctly as a fallback for any specific array implementation.

The `Array{T,n}` type is a specific instance of `AbstractArray` where elements are stored in column-major order. `Vector` and `Matrix` are aliases for the 1-d and 2-d cases. Specific operations such as scalar indexing, assignment, and a few other basic storage-specific operations are all that have to be implemented for `Array`, so that the rest of the array library can be implemented in a generic manner for `AbstractArray`.

`SubArray` is a specialization of `AbstractArray` that performs indexing by reference rather than by copying. A `SubArray` is created with the `sub` function, which is called the same way as `getindex` (with an array and a series of index arguments). The result of `sub` looks the same as the result of `getindex`, except the data is left in place. `sub` stores the input index vectors in a `SubArray` object, which can later be used to index the original array indirectly.

`StridedVector` and `StridedMatrix` are convenient aliases defined to make it possible for Julia to call a wider range of BLAS and LAPACK functions by passing them either `Array` or `SubArray` objects, and thus saving inefficiencies from indexing and memory allocation.

The following example computes the QR decomposition of a small section of a larger array, without creating any temporaries, and by calling the appropriate LAPACK function with the right leading dimension size and stride parameters.

```
julia> a = rand(10,10)
10x10 Float64 Array:
 0.763921  0.884854   0.818783   0.519682   ...  0.860332  0.882295   0.420202
 0.190079  0.235315   0.0669517  0.020172       0.902405  0.0024219  0.24984
 0.823817  0.0285394  0.390379   0.202234       0.516727  0.247442   0.308572
 0.566851  0.622764   0.0683611  0.372167       0.280587  0.227102   0.145647
 0.151173  0.179177   0.0510514  0.615746       0.322073  0.245435   0.976068
 0.534307  0.493124   0.796481   0.0314695  ...  0.843201  0.53461    0.910584
 0.885078  0.891022   0.691548   0.547          0.727538  0.0218296  0.174351
 0.123628  0.833214   0.0224507  0.806369       0.80163   0.457005   0.226993
 0.362621  0.389317   0.702764   0.385856       0.155392  0.497805   0.430512
 0.504046  0.532631   0.477461   0.225632       0.919701  0.0453513  0.505329

julia> b = sub(a, 2:2:8,2:2:4)
4x2 SubArray of 10x10 Float64 Array:
 0.235315  0.020172
 0.622764  0.372167
 0.493124  0.0314695
 0.833214  0.806369

julia> (q,r) = qr(b);

julia> q
4x2 Float64 Array:
 -0.200268   0.331205
 -0.530012   0.107555
 -0.41968    0.720129
 -0.709119  -0.600124
```

```
julia> r
2x2 Float64 Array:
 -1.175  -0.786311
  0.0    -0.414549
```

1.17 Sparse Matrices

Sparse matrices are matrices that contain enough zeros that storing them in a special data structure leads to savings in space and execution time. Sparse matrices may be used when operations on the sparse representation of a matrix lead to considerable gains in either time or space when compared to performing the same operations on a dense matrix.

1.17.1 Compressed Sparse Column (CSC) Storage

In julia, sparse matrices are stored in the Compressed Sparse Column (CSC) format. Julia sparse matrices have the type `SparseMatrixCSC{Tv,Ti}`, where `Tv` is the type of the nonzero values, and `Ti` is the integer type for storing column pointers and row indices.

```
type SparseMatrixCSC{Tv,Ti<:Integer} <: AbstractSparseMatrix{Tv,Ti}
    m::Int                  # Number of rows
    n::Int                  # Number of columns
    colptr::Vector{Ti}      # Column i is in colptr[i]:(colptr[i+1]-1)
    rowval::Vector{Ti}      # Row values of nonzeros
    nzval::Vector{Tv}       # Nonzero values
end
```

The compressed sparse column storage makes it easy and quick to access the elements in the column of a sparse matrix, whereas accessing the sparse matrix by rows is considerably slower. Operations such as insertion of nonzero values one at a time in the CSC structure tend to be slow. This is because all elements of the sparse matrix that are beyond the point of insertion have to be moved one place over.

All operations on sparse matrices are carefully implemented to exploit the CSC data structure for performance, and to avoid expensive operations.

1.17.2 Sparse matrix constructors

The simplest way to create sparse matrices are using functions equivalent to the `zeros` and `eye` functions that Julia provides for working with dense matrices. To produce sparse matrices instead, you can use the same names with an `sp` prefix:

```
julia> spzeros(3,5)
3x5 sparse matrix with 0 nonzeros:

julia> speye(3,5)
3x5 sparse matrix with 3 nonzeros:
    [1, 1]  =  1.0
    [2, 2]  =  1.0
    [3, 3]  =  1.0
```

The `sparse` function is often a handy way to construct sparse matrices. It takes as its input a vector `I` of row indices, a vector `J` of column indices, and a vector `V` of nonzero values. `sparse(I,J,V)` constructs a sparse matrix such that `S[I[k], J[k]] = V[k]`.

```
julia> I = [1, 4, 3, 5]; J = [4, 7, 18, 9]; V = [1, 2, -5, 3];

julia> sparse(I,J,V)
5x18 sparse matrix with 4 nonzeros:
     [1 ,  4]  =  1
     [4 ,  7]  =  2
     [5 ,  9]  =  3
     [3 , 18]  =  -5
```

The inverse of the `sparse` function is `findn`, which retrieves the inputs used to create the sparse matrix.

```
julia> findn(S)
([1, 4, 5, 3],[4, 7, 9, 18])

julia> findn_nzs(S)
([1, 4, 5, 3],[4, 7, 9, 18],[1, 2, 3, -5])
```

Another way to create sparse matrices is to convert a dense matrix into a sparse matrix using the `sparse` function:

```
julia> sparse(eye(5))
5x5 sparse matrix with 5 nonzeros:
    [1, 1]  =  1.0
    [2, 2]  =  1.0
    [3, 3]  =  1.0
    [4, 4]  =  1.0
    [5, 5]  =  1.0
```

You can go in the other direction using the `dense` or the `full` function. The `issparse` function can be used to query if a matrix is sparse.

```
julia> issparse(speye(5))
true
```

1.17.3 Sparse matrix operations

Arithmetic operations on sparse matrices also work as they do on dense matrices. Indexing of, assignment into, and concatenation of sparse matrices work in the same way as dense matrices. Indexing operations, especially assignment, are expensive, when carried out one element at a time. In many cases it may be better to convert the sparse matrix into `(I,J,V)` format using `find_nzs`, manipulate the nonzeros or the structure in the dense vectors `(I,J,V)`, and then reconstruct the sparse matrix.

1.18 Parallel Computing

Most modern computers possess more than one CPU, and several computers can be combined together in a cluster. Harnessing the power of these multiple CPUs allows many computations to be completed more quickly. There are two major factors that influence performance: the speed of the CPUs themselves, and the speed of their access to memory. In a cluster, it's fairly obvious that a given CPU will have fastest access to the RAM within the same computer (node). Perhaps more surprisingly, similar issues are very relevant on a typical multicore laptop, due to differences in the speed of main memory and the cache. Consequently, a good multiprocessing environment should allow control over the "ownership" of a chunk of memory by a particular CPU. Julia provides a multiprocessing environment based on message passing to allow programs to run on multiple processors in separate memory domains at once.

Julia's implementation of message passing is different from other environments such as MPI. Communication in Julia is generally "one-sided", meaning that the programmer needs to explicitly manage only one processor in a two-

processor operation. Furthermore, these operations typically do not look like "message send" and "message receive" but rather resemble higher-level operations like calls to user functions.

Parallel programming in Julia is built on two primitives: *remote references* and *remote calls*. A remote reference is an object that can be used from any processor to refer to an object stored on a particular processor. A remote call is a request by one processor to call a certain function on certain arguments on another (possibly the same) processor. A remote call returns a remote reference to its result. Remote calls return immediately; the processor that made the call proceeds to its next operation while the remote call happens somewhere else. You can wait for a remote call to finish by calling `wait` on its remote reference, and you can obtain the full value of the result using `fetch`.

Let's try this out. Starting with `julia -p n` provides `n` processors on the local machine. Generally it makes sense for `n` to equal the number of CPU cores on the machine.

```
$ ./julia -p 2

julia> r = remote_call(2, rand, 2, 2)
RemoteRef(2,1,5)

julia> fetch(r)
2x2 Float64 Array:
 0.60401   0.501111
 0.174572  0.157411

julia> s = @spawnat 2 1+fetch(r)
RemoteRef(2,1,7)

julia> fetch(s)
2x2 Float64 Array:
 1.60401  1.50111
 1.17457  1.15741
```

The first argument to `remote_call` is the index of the processor that will do the work. Most parallel programming in Julia does not reference specific processors or the number of processors available, but `remote_call` is considered a low-level interface providing finer control. The second argument to `remote_call` is the function to call, and the remaining arguments will be passed to this function. As you can see, in the first line we asked processor 2 to construct a 2-by-2 random matrix, and in the second line we asked it to add 1 to it. The result of both calculations is available in the two remote references, `r` and `s`. The `@spawnat` macro evaluates the expression in the second argument on the processor specified by the first argument.

Occasionally you might want a remotely-computed value immediately. This typically happens when you read from a remote object to obtain data needed by the next local operation. The function `remote_call_fetch` exists for this purpose. It is equivalent to `fetch(remote_call(...))` but is more efficient.

```
julia> remote_call_fetch(2, getindex, r, 1, 1)
0.10824216411304866
```

Remember that `getindex(r,1,1)` is *equivalent* to `r[1,1]`, so this call fetches the first element of the remote reference `r`.

The syntax of `remote_call` is not especially convenient. The macro `@spawn` makes things easier. It operates on an expression rather than a function, and picks where to do the operation for you:

```
julia> r = @spawn rand(2,2)
RemoteRef(1,1,0)

julia> s = @spawn 1+fetch(r)
RemoteRef(1,1,1)

julia> fetch(s)
```

```
1.10824216411304866 1.13798233877923116
1.12376292706355074 1.18750497916607167
```

Note that we used `1+fetch(r)` instead of `1+r`. This is because we do not know where the code will run, so in general a `fetch` might be required to move `r` to the processor doing the addition. In this case, `@spawn` is smart enough to perform the computation on the processor that owns `r`, so the `fetch` will be a no-op.

(It is worth noting that `@spawn` is not built-in but defined in Julia as a *macro*. It is possible to define your own such constructs.)

One important point is that your code must be available on any processor that runs it. For example, type the following into the julia prompt:

```
julia> function rand2(dims...)
         return 2*rand(dims...)
       end

julia> rand2(2,2)
2x2 Float64 Array:
 0.153756  0.368514
 1.15119   0.918912

julia> @spawn rand2(2,2)
RemoteRef(1,1,1)

julia> @spawn rand2(2,2)
RemoteRef(2,1,2)

julia> exception on 2: in anonymous: rand2 not defined
```

Processor 1 knew about the function `rand2`, but processor 2 did not. To make your code available to all processors, the `require` function will automatically load a source file on all currently available processors:

```
julia> require("myfile")
```

In a cluster, the contents of the file (and any files loaded recursively) will be sent over the network.

1.18.1 Data Movement

Sending messages and moving data constitute most of the overhead in a parallel program. Reducing the number of messages and the amount of data sent is critical to achieving performance and scalability. To this end, it is important to understand the data movement performed by Julia's various parallel programming constructs.

`fetch` can be considered an explicit data movement operation, since it directly asks that an object be moved to the local machine. `@spawn` (and a few related constructs) also moves data, but this is not as obvious, hence it can be called an implicit data movement operation. Consider these two approaches to constructing and squaring a random matrix:

```
# method 1
A = rand(1000,1000)
Bref = @spawn A^2
...
fetch(Bref)

# method 2
Bref = @spawn rand(1000,1000)^2
...
fetch(Bref)
```

The difference seems trivial, but in fact is quite significant due to the behavior of `@spawn`. In the first method, a random matrix is constructed locally, then sent to another processor where it is squared. In the second method, a random matrix is both constructed and squared on another processor. Therefore the second method sends much less data than the first.

In this toy example, the two methods are easy to distinguish and choose from. However, in a real program designing data movement might require more thought and very likely some measurement. For example, if the first processor needs matrix `A` then the first method might be better. Or, if computing `A` is expensive and only the current processor has it, then moving it to another processor might be unavoidable. Or, if the current processor has very little to do between the `@spawn` and `fetch(Bref)` then it might be better to eliminate the parallelism altogether. Or imagine `rand(1000,1000)` is replaced with a more expensive operation. Then it might make sense to add another `@spawn` statement just for this step.

1.18.2 Parallel Map and Loops

Fortunately, many useful parallel computations do not require data movement. A common example is a monte carlo simulation, where multiple processors can handle independent simulation trials simultaneously. We can use `@spawn` to flip coins on two processors. First, write the following function in `count_heads.jl`:

```
function count_heads(n)
    c::Int = 0
    for i=1:n
        c += randbool()
    end
    c
end
```

The function `count_heads` simply adds together `n` random bits. Here is how we can perform some trials on two machines, and add together the results:

```
require("count_heads")

a = @spawn count_heads(100000000)
b = @spawn count_heads(100000000)
fetch(a)+fetch(b)
```

This example, as simple as it is, demonstrates a powerful and often-used parallel programming pattern. Many iterations run independently over several processors, and then their results are combined using some function. The combination process is called a *reduction*, since it is generally tensor-rank-reducing: a vector of numbers is reduced to a single number, or a matrix is reduced to a single row or column, etc. In code, this typically looks like the pattern `x = f(x,v[i])`, where `x` is the accumulator, `f` is the reduction function, and the `v[i]` are the elements being reduced. It is desirable for `f` to be associative, so that it does not matter what order the operations are performed in.

Notice that our use of this pattern with `count_heads` can be generalized. We used two explicit `@spawn` statements, which limits the parallelism to two processors. To run on any number of processors, we can use a *parallel for loop*, which can be written in Julia like this:

```
nheads = @parallel (+) for i=1:200000000
  randbool()
end
```

This construct implements the pattern of assigning iterations to multiple processors, and combining them with a specified reduction (in this case `(+)`). The result of each iteration is taken as the value of the last expression inside the loop. The whole parallel loop expression itself evaluates to the final answer.

Note that although parallel for loops look like serial for loops, their behavior is dramatically different. In particular, the iterations do not happen in a specified order, and writes to variables or arrays will not be globally visible since

iterations run on different processors. Any variables used inside the parallel loop will be copied and broadcast to each processor.

For example, the following code will not work as intended:

```
a = zeros(100000)
@parallel for i=1:100000
  a[i] = i
end
```

Notice that the reduction operator can be omitted if it is not needed. However, this code will not initialize all of `a`, since each processor will have a separate copy if it. Parallel for loops like these must be avoided. Fortunately, distributed arrays can be used to get around this limitation, as we will see in the next section.

Using "outside" variables in parallel loops is perfectly reasonable if the variables are read-only:

```
a = randn(1000)
@parallel (+) for i=1:100000
  f(a[randi(end)])
end
```

Here each iteration applies `f` to a randomly-chosen sample from a vector `a` shared by all processors.

In some cases no reduction operator is needed, and we merely wish to apply a function to all integers in some range (or, more generally, to all elements in some collection). This is another useful operation called *parallel map*, implemented in Julia as the `pmap` function. For example, we could compute the singular values of several large random matrices in parallel as follows:

```
M = {rand(1000,1000) for i=1:10}
pmap(svd, M)
```

Julia's `pmap` is designed for the case where each function call does a large amount of work. In contrast, `@parallel for` can handle situations where each iteration is tiny, perhaps merely summing two numbers.

1.18.3 Synchronization With Remote References

1.18.4 Scheduling

Julia's parallel programming platform uses *Tasks (aka Coroutines)* to switch among multiple computations. Whenever code performs a communication operation like `fetch` or `wait`, the current task is suspended and a scheduler picks another task to run. A task is restarted when the event it is waiting for completes.

For many problems, it is not necessary to think about tasks directly. However, they can be used to wait for multiple events at the same time, which provides for *dynamic scheduling*. In dynamic scheduling, a program decides what to compute or where to compute it based on when other jobs finish. This is needed for unpredictable or unbalanced workloads, where we want to assign more work to processors only when they finish their current tasks.

As an example, consider computing the singular values of matrices of different sizes:

```
M = {rand(800,800), rand(600,600), rand(800,800), rand(600,600)}
pmap(svd, M)
```

If one processor handles both 800x800 matrices and another handles both 600x600 matrices, we will not get as much scalability as we could. The solution is to make a local task to "feed" work to each processor when it completes its current task. This can be seen in the implementation of `pmap`:

```
function pmap(f, lst)
    np = nprocs()  # determine the number of processors available
    n = length(lst)
```

```
    results = cell(n)
    i = 1
    # function to produce the next work item from the queue.
    # in this case it's just an index.
    next_idx() = (idx=i; i+=1; idx)
    @sync begin
        for p=1:np
            @spawnlocal begin
                while true
                    idx = next_idx()
                    if idx > n
                        break
                    end
                    results[idx] = remote_call_fetch(p, f, lst[idx])
                end
            end
        end
    end
    results
end
```

`@spawnlocal` is similar to `@spawn`, but only runs tasks on the local processor. We use it to create a "feeder" task for each processor. Each task picks the next index that needs to be computed, then waits for its processor to finish, then repeats until we run out of indexes. A `@sync` block is used to wait for all the local tasks to complete, at which point the whole operation is done. Notice that all the feeder tasks are able to share state via `next_idx()` since they all run on the same processor. However, no locking is required, since the threads are scheduled cooperatively and not preemptively. This means context switches only occur at well-defined points (during the `fetch` operation).

1.18.5 Sending Instructions To All Processors

It is often useful to execute a statement on all processors, particularly for setup tasks such as loading source files and defining common variables. This can be done with the `@everywhere` macro:

@everywhere include("defs.jl")

1.19 Running External Programs

Julia borrows backtick notation for commands from the shell, Perl, and Ruby. However, in Julia, writing

```
julia> `echo hello`
`echo hello`
```

differs in a several aspects from the behavior in various shells, Perl, or Ruby:

- Instead of immediately running the command, backticks create a `Cmd` object to represent the command. You can use this object to connect the command to others via pipes, run it, and read or write to it.
- When the command is run, Julia does not capture its output unless you specifically arrange for it to. Instead, the output of the command by default goes to `stdout` as it would using `libc`'s `system` call.
- The command is never run with a shell. Instead, Julia parses the command syntax directly, appropriately interpolating variables and splitting on words as the shell would, respecting shell quoting syntax. The command is run as `julia`'s immediate child process, using `fork` and `exec` calls.

Here's a simple example of actually running an external program:

```
julia> run(`echo hello`)
hello
true
```

The `hello` is the output of the `echo` command, sent to stdout. The run method itself returns `Nothing`, and throws an `ErrorException` if the external command fails to run successfully.

If you want to read the output of the external command, the `readall` method can be used instead:

```
julia> a=readall(`echo hello`)
"hello\n"

julia> (chomp(a)) == "hello"
true
```

1.19.1 Interpolation

Suppose you want to do something a bit more complicated and use the name of a file in the variable `file` as an argument to a command. You can use `$` for interpolation much as you would in a string literal (see *Strings*):

```
julia> file = "/etc/passwd"
"/etc/passwd"

julia> `sort $file`
`sort /etc/passwd`
```

A common pitfall when running external programs via a shell is that if a file name contains characters that are special to the shell, they may cause undesirable behavior. Suppose, for example, rather than `/etc/passwd`, we wanted to sort the contents of the file `/Volumes/External HD/data.csv`. Let's try it:

```
julia> file = "/Volumes/External HD/data.csv"
"/Volumes/External HD/data.csv"

julia> `sort $file`
`sort '/Volumes/External HD/data.csv'`
```

How did the file name get quoted? Julia knows that `file` is meant to be interpolated as a single argument, so it quotes the word for you. Actually, that is not quite accurate: the value of `file` is never interpreted by a shell, so there's no need for actual quoting; the quotes are inserted only for presentation to the user. This will even work if you interpolate a value as part of a shell word:

```
julia> path = "/Volumes/External HD"
"/Volumes/External HD"

julia> name = "data"
"data"

julia> ext = "csv"
"csv"

julia> `sort $path/$name.$ext`
`sort '/Volumes/External HD/data.csv'`
```

As you can see, the space in the `path` variable is appropriately escaped. But what if you *want* to interpolate multiple words? In that case, just use an array (or any other iterable container):

```
julia> files = ["/etc/passwd","/Volumes/External HD/data.csv"]
2-element ASCIIString Array:
```

```
 "/etc/passwd"
 "/Volumes/External HD/data.csv"


julia> `grep foo $files`
`grep foo /etc/passwd '/Volumes/External HD/data.csv'`
```

If you interpolate an array as part of a shell word, Julia emulates the shell's `{a,b,c}` argument generation:

```
julia> names = ["foo","bar","baz"]
3-element ASCIIString Array:
 "foo"
 "bar"
 "baz"

julia> `grep xylophone $names.txt`
`grep xylophone foo.txt bar.txt baz.txt`
```

Moreover, if you interpolate multiple arrays into the same word, the shell's Cartesian product generation behavior is emulated:

```
julia> names = ["foo","bar","baz"]
3-element ASCIIString Array:
 "foo"
 "bar"
 "baz"

julia> exts = ["aux","log"]
2-element ASCIIString Array:
 "aux"
 "log"

julia> `rm -f $names.$exts`
`rm -f foo.aux foo.log bar.aux bar.log baz.aux baz.log`
```

Since you can interpolate literal arrays, you can use this generative functionality without needing to create temporary array objects first:

```
julia> `rm -rf $["foo","bar","baz","qux"].$["aux","log","pdf"]`
`rm -rf foo.aux foo.log foo.pdf bar.aux bar.log bar.pdf baz.aux baz.log baz.pdf qux.aux qux.log qux.p
```

1.19.2 Quoting

Inevitably, one wants to write commands that aren't quite so simple, and it becomes necessary to use quotes. Here's a simple example of a perl one-liner at a shell prompt:

```
sh$ perl -le '$|=1; for (0..3) { print }'
0
1
2
3
```

The Perl expression needs to be in single quotes for two reasons: so that spaces don't break the expression into multiple shell words, and so that uses of Perl variables like `$|` (yes, that's the name of a variable in Perl), don't cause interpolation. In other instances, you may want to use double quotes so that interpolation *does* occur:

```
sh$ first="A"
sh$ second="B"
sh$ perl -le '$|=1; print for @ARGV' "1: $first" "2: $second"
1: A
2: B
```

In general, the Julia backtick syntax is carefully designed so that you can just cut-and-paste shell commands as-is into backticks and they will work: the escaping, quoting, and interpolation behaviors are the same as the shell's. The only difference is that the interpolation is integrated and aware of Julia's notion of what is a single string value, and what is a container for multiple values. Let's try the above two examples in Julia:

```
julia> `perl -le '$|=1; for (0..3) { print }'`
`perl -le '$|=1; for (0..3) { print }'`

julia> run(ans)
0
1
2
3
true

julia> first = "A"; second = "B";

julia> `perl -le 'print for @ARGV' "1: $first" "2: $second"`
`perl -le 'print for @ARGV' '1: A' '2: B'`

julia> run(ans)
1: A
2: B
true
```

The results are identical, and Julia's interpolation behavior mimics the shell's with some improvements due to the fact that Julia supports first-class iterable objects while most shells use strings split on spaces for this, which introduces ambiguities. When trying to port shell commands to Julia, try cut and pasting first. Since Julia shows commands to you before running them, you can easily and safely just examine its interpretation without doing any damage.

1.19.3 Pipelines

Shell metacharacters, such as `|`, `&`, and >, are not special inside of Julia's backticks: unlike in the shell, inside of Julia's backticks, a pipe is always just a pipe:

```
julia> run(`echo hello | sort`)
hello | sort
true
```

This expression invokes the `echo` command with three words as arguments: "hello", "|", and "sort". The result is that a single line is printed: "hello | sort". Inside of backticks, a "|" is just a literal pipe character. How, then, does one construct a pipeline? Instead of using "|" inside of backticks, one uses Julia's `|` operator between `Cmd` objects:

```
julia> run(`echo hello` | `sort`)
hello
true
```

This pipes the output of the `echo` command to the `sort` command. Of course, this isn't terribly interesting since there's only one line to sort, but we can certainly do much more interesting things:

```
julia> run(`cut -d: -f3 /etc/passwd` | `sort -n` | `tail -n5`)
210
211
212
213
214
true
```

This prints the highest five user IDs on a UNIX system. The `cut`, `sort` and `tail` commands are all spawned as immediate children of the current `julia` process, with no intervening shell process. Julia itself does the work to setup pipes and connect file descriptors that is normally done by the shell. Since Julia does this itself, it retains better control and can do some things that shells cannot.

Julia can run multiple commands in parallel:

```
julia> run(`echo hello` & `echo world`)
world
hello
true
```

The order of the output here is non-deterministic because the two `echo` processes are started nearly simultaneously, and race to make the first write to the `stdout` descriptor they share with each other and the `julia` parent process. Julia lets you pipe the output from both of these processes to another program:

```
julia> run(`echo world` & `echo hello` | `sort`)
hello
world
true
```

In terms of UNIX plumbing, what's happening here is that a single UNIX pipe object is created and written to by both `echo` processes, and the other end of the pipe is read from by the `sort` command.

The combination of a high-level programming language, a first-class command abstraction, and automatic setup of pipes between processes is a powerful one. To give some sense of the complex pipelines that can be created easily, here are some more sophisticated examples, with apologies for the excessive use of Perl one-liners:

```
julia> prefixer(prefix, sleep) = `perl -nle '$|=1; print "'$prefix' ", $_; sleep '$sleep';'`

julia> run(`perl -le '$|=1; for(0..9){ print; sleep 1 }'` | prefixer("A",2) & prefixer("B",2))
A   0
B   1
A   2
B   3
A   4
B   5
A   6
B   7
A   8
B   9
true
```

This is a classic example of a single producer feeding two concurrent consumers: one `perl` process generates lines with the numbers 0 through 9 on them, while two parallel processes consume that output, one prefixing lines with the letter "A", the other with the letter "B". Which consumer gets the first line is non-deterministic, but once that race has been won, the lines are consumed alternately by one process and then the other. (Setting `$|=1` in Perl causes each print statement to flush the `stdout` handle, which is necessary for this example to work. Otherwise all the output is buffered and printed to the pipe at once, to be read by just one consumer process.)

Here is an even more complex multi-stage producer-consumer example:

```
julia> run(`perl -le '$|=1; for(0..9){ print; sleep 1 }'` |
           prefixer("X",3) & prefixer("Y",3) & prefixer("Z",3) |
           prefixer("A",2) & prefixer("B",2))
B   Y   0
A   Z   1
B   X   2
A   Y   3
B   Z   4
A   X   5
B   Y   6
A   Z   7
B   X   8
A   Y   9
true
```

This example is similar to the previous one, except there are two stages of consumers, and the stages have different latency so they use a different number of parallel workers, to maintain saturated throughput.

Finally, we have an example of how you can make a process read from itself:

```
julia> gen = `perl -le '$|=1; for(0..9){ print; sleep 1 }'`
`perl -le '$|=1; for(0..9){ print; sleep 1 }'`

julia> dup = `perl -ne '$|=1; warn $_; print ".$_"; sleep 1'`
`perl -ne '$|=1; warn $_; print ".$_"; sleep 1'`

julia> run(gen | dup | dup)
0
.0
1
..0
2
.1
3
...0
4
.2
5
..1
6
.3
....0
7
.4
8
9
..2
.5
...1
.6
..3
.....0
.7
..4
.8
.9
...2
..5
....1
```

```
..6
...3
```

This example never terminates since the `dup` process reads its own output and duplicates it to `stderr` forever. We strongly encourage you to try all these examples to see how they work.

1.20 Calling C and Fortran Code

Though most code can be written in Julia, there are many high-quality, mature libraries for numerical computing already written in C and Fortran. To allow easy use of this existing code, Julia makes it simple and efficient to call C and Fortran functions. Julia has a "no boilerplate" philosophy: functions can be called directly from Julia without any "glue" code, code generation, or compilation — even from the interactive prompt. This is accomplished just by making an appropriate call with `call` syntax, which looks like an ordinary function call.

The code to be called must be available as a shared library. Most C and Fortran libraries ship compiled as shared libraries already, but if you are compiling the code yourself using GCC (or Clang), you will need to use the `-shared` and `-fPIC` options. The machine instructions generated by Julia's JIT are the same as a native C call would be, so the resulting overhead is the same as calling a library function from C code. (Non-library function calls in both C and Julia can be inlined and thus may have even less overhead than calls to shared library functions. When both libraries and executables are generated by LLVM, it is possible to perform whole-program optimizations that can even optimize across this boundary, but Julia does not yet support that. In the future, however, it may do so, yielding even greater performance gains.)

Shared libraries and functions are referenced by a tuple of the form `(:function, "library")` or `("function", "library")` where `function` is the C-exported function name. `library` refers to the shared library name: shared libraries available in the (platform-specific) load path will be resolved by name, and if necessary a direct path may be specified.

A function name may be used alone in place of the tuple (just `:function` or `"function"`). In this case the name is resolved within the current process. This form can be used to call C library functions, functions in the Julia runtime, or functions in an application linked to Julia.

Finally, you can use `ccall` to actually generate a call to the library function. Arguments to `ccall` are as follows:

1. (:function, "library") pair (must be a constant, but see below).
2. Return type, which may be any bits type, including `Int32`, `Int64`, `Float64`, or `Ptr{T}` for any type parameter `T`, indicating a pointer to values of type `T`, or just `Ptr` for `void*` "untyped pointer" values.
3. A tuple of input types, like those allowed for the return type.
4. The following arguments, if any, are the actual argument values passed to the function.

As a complete but simple example, the following calls the `clock` function from the standard C library:

```
julia> t = ccall( (:clock, "libc"), Int32, ())
2292761

julia> t
2292761

julia> typeof(ans)
Int32
```

`clock` takes no arguments and returns an `Int32`. One common gotcha is that a 1-tuple must be written with with a trailing comma. For example, to call the `getenv` function to get a pointer to the value of an environment variable, one makes a call like this:

```
julia> path = ccall( (:getenv, "libc"), Ptr{Uint8}, (Ptr{Uint8},), "SHELL")
Ptr{Uint8} @0x00007fff5fbffc45

julia> bytestring(path)
"/bin/bash"
```

Note that the argument type tuple must be written as `(Ptr{Uint8},)`, rather than `(Ptr{Uint8})`. This is because `(Ptr{Uint8})` is just `Ptr{Uint8}`, rather than a 1-tuple containing `Ptr{Uint8}`:

```
julia> (Ptr{Uint8})
Ptr{Uint8}

julia> (Ptr{Uint8},)
(Ptr{Uint8},)
```

In practice, especially when providing reusable functionality, one generally wraps `ccall` uses in Julia functions that set up arguments and then check for errors in whatever manner the C or Fortran function indicates them, propagating to the Julia caller as exceptions. This is especially important since C and Fortran APIs are notoriously inconsistent about how they indicate error conditions. For example, the `getenv` C library function is wrapped in the following Julia function in env.jl:

```
function getenv(var::String)
  val = ccall( (:getenv, "libc"),
               Ptr{Uint8}, (Ptr{Uint8},), bytestring(var))
  if val == C_NULL
    error("getenv: undefined variable: ", var)
  end
  bytestring(val)
end
```

The C `getenv` function indicates an error by returning `NULL`, but other standard C functions indicate errors in various different ways, including by returning -1, 0, 1 and other special values. This wrapper throws an exception clearly indicating the problem if the caller tries to get a non-existent environment variable:

```
julia> getenv("SHELL")
"/bin/bash"

julia> getenv("FOOBAR")
getenv: undefined variable: FOOBAR
```

Here is a slightly more complex example that discovers the local machine's hostname:

```
function gethostname()
  hostname = Array(Uint8, 128)
  ccall( (:gethostname, "libc"), Int32,
         (Ptr{Uint8}, Uint),
         hostname, length(hostname))
  return bytestring(convert(Ptr{Uint8}, hostname))
end
```

This example first allocates an array of bytes, then calls the C library function `gethostname` to fill the array in with the hostname, takes a pointer to the hostname buffer, and converts the pointer to a Julia string, assuming that it is a NUL-terminated C string. It is common for C libraries to use this pattern of requiring the caller to allocate memory to be passed to the callee and filled in. Allocation of memory from Julia like this is generally accomplished by creating an uninitialized array and passing a pointer to its data to the C function.

When calling a Fortran function, all inputs must be passed by reference.

A prefix `&` is used to indicate that a pointer to a scalar argument should be passed instead of the scalar value itself. The following example computes a dot product using a BLAS function.

```
function compute_dot(DX::Vector, DY::Vector)
  assert(length(DX) == length(DY))
  n = length(DX)
  incx = incy = 1
  product = ccall( (:ddot_, "libLAPACK"),
                  Float64,
                  (Ptr{Int32}, Ptr{Float64}, Ptr{Int32}, Ptr{Float64}, Ptr{Int32}),
                  &n, DX, &incx, DY, &incy)
  return product
end
```

The meaning of prefix `&` is not quite the same as in C. In particular, any changes to the referenced variables may not be visible in Julia (the goal is to make any changes visible in the spirit of C, but this is not currently implemented for immutable types). However, it will never cause any harm for called functions to attempt such modifications (that is, writing through the passed pointers). Since this `&` is not a real address operator, it may be used with any syntax, such as `&0` or `&f(x)`.

Note that no C header files are used anywhere in the process. Currently, it is not possible to pass structs and other non-primitive types from Julia to C libraries. However, C functions that generate and use opaque structs types by passing around pointers to them can return such values to Julia as `Ptr{Void}`, which can then be passed to other C functions as `Ptr{Void}`. Memory allocation and deallocation of such objects must be handled by calls to the appropriate cleanup routines in the libraries being used, just like in any C program.

1.20.1 Mapping C Types to Julia

Julia automatically inserts calls to the `convert` function to convert each argument to the specified type. For example, the following call:

```
ccall( (:foo, "libfoo"), Void, (Int32, Float64),
      x, y)
```

will behave as if the following were written:

```
ccall( (:foo, "libfoo"), Void, (Int32, Float64),
      convert(Int32, x), convert(Float64, y))
```

When a scalar value is passed with `&` as an argument of type `Ptr{T}`, the value will first be converted to type `T`.

Array conversions

When an `Array` is passed to C as a `Ptr` argument, it is "converted" simply by taking the address of the first element. This is done in order to avoid copying arrays unnecessarily, and to tolerate the slight mismatches in pointer types that are often encountered in C APIs (for example, passing a `Float64` array to a function that operates on uninterpreted bytes).

Therefore, if an `Array` contains data in the wrong format, it will have to be explicitly converted using a call such as `int32(a)`.

Type correspondences

On all systems we currently support, basic C/C++ value types may be translated to Julia types as follows. Every C type also has a corresponding Julia type with the same name, prefixed by C. This can help for writing portable code (and remembering that an int in C is not the same as an Int in Julia).

System-independent:

`bool` (8 bits)	`Cbool`	`Bool`
`signed char`		`Int8`
`unsigned char`	`Cuchar`	`Uint8`
`short`	`Cshort`	`Int16`
`unsigned short`	`Cushort`	`Uint16`
`int`	`Cint`	`Int32`
`unsigned int`	`Cuint`	`Uint32`
`long long`	`Clonglong`	`Int64`
`unsigned long long`	`Culonglong`	`Uint64`
`float`	`Cfloat`	`Float32`
`double`	`Cdouble`	`Float64`
`ptrdiff_t`	`Cptrdiff_t`	`Int`
`size_t`	`Csize_t`	`Uint`
`complex float`	`Ccomplex_float` (future addition)	
`complex double`	`Ccomplex_double` (future addition)	
`void`		`Void`
`void*`		`Ptr{Void}`
`char*` (or `char[]`, e.g. a string)		`Ptr{Uint8}`
`char**` (or `*char[]`)		`Ptr{Ptr{Uint8}}`
`struct T*` (where T represents an appropriately defined bits type)		`Ptr{T}` (call using &variable_name in the parameter list)
`struct T` (where T represents an appropriately defined bits type)		`T` (call using &variable_name in the parameter list)
`jl_value_t*` (any Julia Type)		`Ptr{Any}`

Note: the `bool` type is only defined by C++, where it is 8 bits wide. In C, however, `int` is often used for boolean values. Since `int` is 32-bits wide (on all supported systems), there is some potential for confusion here.

A C function declared to return `void` will give `nothing` in Julia.

System-dependent:

`char`	`Cchar`	`Int8` (x86, x86_64) `Uint8` (powerpc, arm)
`long`	`Clong`	`Int` (UNIX) `Int32` (Windows)
`unsigned long`	`Culong`	`Uint` (UNIX) `Int32` (Windows)
`wchar_t`	`Char`	Although `wchar_t` is technically system-dependent, on all the systems we currently support (UNIX), it is 32-bit.

For string arguments (`char*`) the Julia type should be `Ptr{Uint8}`, not `ASCIIString`. C functions that take an argument of the type `char**` can be called by using a `Ptr{Ptr{Uint8}}` type within Julia. For example, C functions of the form:

```
int main(int argc, char **argv);
```

can be called via the following Julia code:

```
argv = [ "a.out", "arg1", "arg2" ]
ccall(:main, Int32, (Int32, Ptr{Ptr{Uint8}}), length(argv), argv)
```

1.20.2 Accessing Data through a Pointer

The following methods are described as "unsafe" because they can cause Julia to terminate abruptly or corrupt arbitrary process memory due to a bad pointer or type declaration.

Given a `Ptr{T}`, the contents of type `T` can generally be copied from the referenced memory into a Julia type using `unsafe_ref(ptr, [index])`. The index argument is optional (default is 1), and performs 1-based indexing.

This function is intentionally similar to the behavior of `getindex()` and `setindex!()` (e.g. `[]` access syntax).

If T is a bitstype, the return value will be that number.

If T is a type or immutable, the return value will be a new object initialized to contain a copy of the contents of the referenced memory. The referenced memory can safely be freed or released.

If T is Any, then the referenced memory is assumed to contain some `jl_value_t*` and is not copied. You must be careful in this case to ensure that the object was always visible to the garbage collector (pointers do not count, but the new object does) to ensure the memory is not prematurely freed. Note that if the object was not originally allocated by Julia, the new object will never be finalized by Julia's garbage collector. If the `Ptr` itself is actually a `jl_value_t*`, it can be converted back to a Julia object reference by `unsafe_pointer_to_objref(ptr)`. [Julia values `v` can be converted to `jl_value_t*` pointers (`Ptr{Void}`) by calling `pointer_from_objref(v)`.]

The reverse operation (writing data to a Ptr{T}), can be performed using `unsafe_assign(ptr, value, [index])`. Currently, this is only supported for bitstypes or other pointer-free (`isbits`) immutable types.

Any operation that throws an error is probably currently unimplemented and should be posted as a bug so that it can be resolved.

If the pointer of interest is an array of bits (bitstype or immutable), the function `pointer_to_array(ptr,dims,[own])` may be more more useful. The final parameter should be true if Julia should "take ownership" of the underlying buffer and call `free(ptr)` when the returned `Array` object is finalized. If the `own` parameter is omitted or false, the caller must ensure the buffer remains in existence until all access is complete.

1.20.3 Garbage Collection Safety

When passing data to a ccall, it is best to avoid using the `pointer()` function. Instead define a convert method and pass the variables directly to the ccall. ccall automatically arranges that all of its arguments will be preserved from garbage collection until the call returns. If a C API will store a reference to memory allocated by Julia, after the ccall returns, you must arrange that the object remains visible to the garbage collector. The suggested way to handle this is to make a global variable of type `Array{Any,1}` to hold these values, until C interface notifies you that it is finished with them.

Whenever you have created a pointer to Julia data, you must ensure the original data exists until you are done with using the pointer. Many methods in Julia such as `unsafe_ref()` and `bytestring()` make copies of data instead of taking ownership of the buffer, so that it is safe to free (or alter) the original data without affecting Julia. A notable exception is `pointer_to_array()` which, for performance reasons, shares (or can be told to take ownership of) the underlying buffer.

1.20.4 Non-constant Function Specifications

A `(name, library)` function specification must be a constant expression. However, it is possible to use computed values as function names by staging through `eval` as follows:

```
@eval ccall(($(string("a","b")),"lib"), ...
```

This expression constructs a name using `string`, then substitutes this name into a new `ccall` expression, which is then evaluated. Keep in mind that `eval` only operates at the top level, so within this expression local variables will not be available (unless their values are substituted with `$`). For this reason, `eval` is typically only used to form top-level definitions, for example when wrapping libraries that contain many similar functions.

1.20.5 Indirect calls

The first argument to `ccall` can also be an expression evaluated at run time. In this case, the expression must evaluate to a `Ptr`, which will be used as the address of the native function to call. This behavior occurs when the first `ccall` argument contains references to non-constants, such as local variables or function arguments.

1.20.6 C++

Limited support for C++ is provided by the *Cpp* package.

1.21 Julia Packages

1.21.1 Where to find Julia packages

- An official list of packages is available, see *Pacotes Disponíveis (em inglês)*.
- Announcements of new packages can also be found in the julia-users Google Groups.

1.21.2 Installing a new Julia package

The *Pkg* module in julia provides tools for installing and managing third party packages. It also manages the dependencies, while installing packages. Get the updated list of packages with:

```
Pkg.update()
```

In order to install a package, use `Pkg.add()`, where `MY_PACKAGE_NAME` is replaced with the actual package name:

```
Pkg.add("MY_PACKAGE_NAME")
```

This installs the package to `$HOME/.julia/MY_PACKAGE_NAME` . In order to remove a package, do:

```
Pkg.rm("MY_PACKAGE_NAME")
```

Internally, every Julia package is a `git` repository, and Julia uses `git` for its package management.

1.21.3 Contributing a new Julia package

In the following, replace `MY_PACKAGE_NAME`, `MY_GITHUB_USER`, etc. with the actual desired names.

Creating a new Julia package

1. Initialize your package in Julia by running:

   ```
   Pkg.new("MY_PACKAGE_NAME")
   ```

This will initialize a skeleton for a new package in `$HOME/.julia/MY_PACKAGE_NAME`.

Nota: This will overwrite any existing files and git repository in `$HOME/.julia/MY_PACKAGE_NAME`.

2. If you have already created a repository for your package, overwrite the skeleton by copying or symlinking over it. For example:

```
rm -r $HOME/.julia/MY_PACKAGE_NAME
ln -s /path/to/existing/repo/MY_PACKAGE_NAME $HOME/.julia/MY_PACKAGE_NAME
```

3. In `REQUIRE`, list the names of all packages used by your new package. One package per line.

4. Populate the package by filling out `README.md` and `LICENSE.md`, source code in `src/`, and tests in `test/`. Ensure that each test file contains these lines near the beginning:

```
using Test
using MY_PACKAGE_NAME
```

5. Add a publicly accessible remote repository URL, if your package doesn't already have one. For example, create a new repository called `MY_PACKAGE_NAME.jl` on Github and then run:

```
cd $HOME/.julia/MY_PACKAGE_NAME
git remote add github https://github.com/MY_GITHUB_USER/MY_PACKAGE_NAME.jl
```

6. Add at least one git commit and push it to the remote repository.

```
# Do some stuff
git add #new files
git commit
git push remote github
```

Distributing a Julia package

1.21.4 One-time setup (once per user)

1. Fork a copy of METADATA.jl, if you haven't done so already. The forked repository URL should look like *https://github.com/MY_GITHUB_USER/METADATA.jl*.

2. Update the local METADATA with the URL of your forked repository.:

```
cd $HOME/.julia/METADATA
git remote add github https://github.com/MY_GITHUB_USER/METADATA.jl
```

1.21.5 Distributing a new package or new version of an existing package

1. Populate the local METADATA by running in Julia:

```
Pkg.pkg_origin("MY_PACKAGE_NAME")
Pkg.patch("MY_PACKAGE_NAME")
```

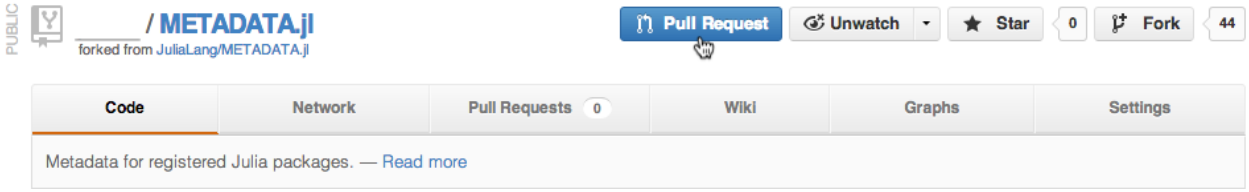
2. Update the local METADATA with the URL of your forked repository and create a new branch with your package in it.

```
cd $HOME/.julia/METADATA
git branch MY_PACKAGE_NAME
git checkout MY_PACKAGE_NAME
git add MY_PACKAGE_NAME #Ensure that only the latest hash is committed
git commit
```

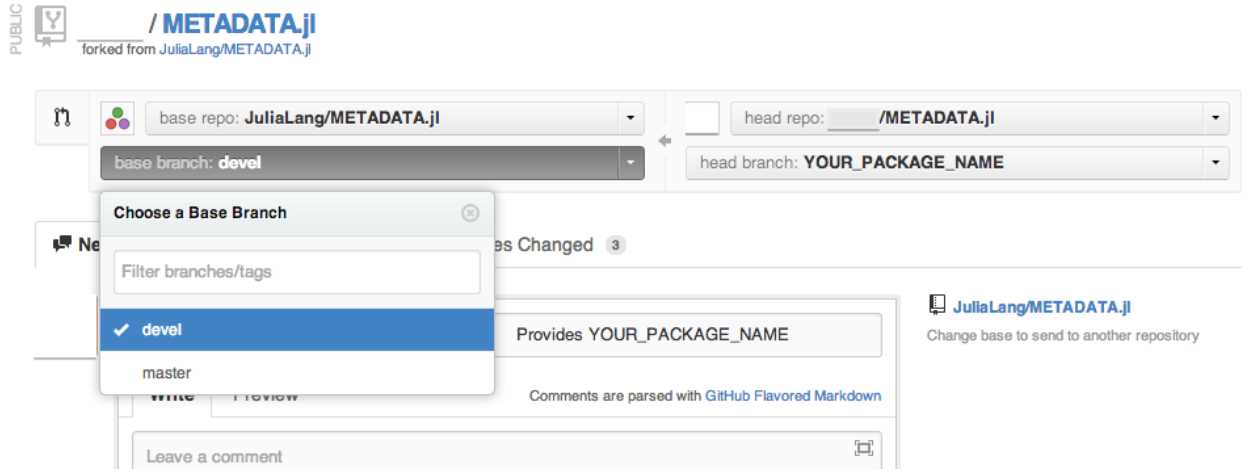
3. Push to the remote METADATA repository:

```
git push github MY_PACKAGE_NAME
```

4. Go to *https://github.com/MY_GITHUB_USER/METADATA.jl/tree/MY_PACKAGE_NAME* in your web browser. Click the 'Pull Request' button.



5. Submit a new pull request. Ensure that the pull request goes to the devel branch and not master.



6. When the pull request is accepted, announce your new package to the Julia community on the julia-users Google Groups.

1.22 Performance Tips

In the following sections, we briefly go through a few techniques that can help make your Julia code run as fast as possible.

1.22.1 Avoid global variables

A global variable might have its value, and therefore its type, change at any point. This makes it difficult for the compiler to optimize code using global variables. Variables should be local, or passed as arguments to functions, whenever possible.

We find that global names are frequently constants, and declaring them as such greatly improves performance:

```
const DEFAULT_VAL = 0
```

Uses of non-constant globals can be optimized by annotating their types at the point of use:

```
global x
y = f(x::Int + 1)
```

1.22.2 Type declarations

In many languages with optional type declarations, adding declarations is the principal way to make code run faster. In Julia, the compiler generally knows the types of all function arguments and local variables. However, there are a

few specific instances where declarations are helpful.

Declare specific types for fields of composite types

Given a user-defined type like the following:

```
type Foo
    field
end
```

the compiler will not generally know the type of `foo.field`, since it might be modified at any time to refer to a value of a different type. It will help to declare the most specific type possible, such as `field::Float64` or `field::Array{Int64,1}`.

Annotate values taken from untyped locations

It is often convenient to work with data structures that may contain values of any type, such as the original `Foo` type above, or cell arrays (arrays of type `Array{Any}`). But, if you're using one of these structures and happen to know the type of an element, it helps to share this knowledge with the compiler:

```
function foo(a::Array{Any,1})
    x = a[1]::Int32
    b = x+1
    ...
end
```

Here, we happened to know that the first element of `a` would be an `Int32`. Making an annotation like this has the added benefit that it will raise a run-time error if the value is not of the expected type, potentially catching certain bugs earlier.

Declare types of named arguments

Named arguments can have declared types:

```
function with_named(x; name::Int = 1)
    ...
end
```

Functions are specialized on the types of named arguments, so these declarations will not affect performance of code inside the function. However, they will reduce the overhead of calls to the function that include named arguments.

Functions with named arguments have near-zero overhead for call sites that pass only positional arguments.

Passing dynamic lists of named arguments, as in `f(x; names...)`, can be slow and should be avoided in performance-sensitive code.

1.22.3 Break functions into multiple definitions

Writing a function as many small definitions allows the compiler to directly call the most applicable code, or even inline it.

Here is an example of a "compound function" that should really be written as multiple definitions:

```
function norm(A)
    if isa(A, Vector)
        return sqrt(real(dot(x,x)))
    elseif isa(A, Matrix)
        return max(svd(A)[2])
    else
        error("norm: invalid argument")
    end
end
```

This can be written more concisely and efficiently as:

```
norm(A::Vector) = sqrt(real(dot(x,x)))
norm(A::Matrix) = max(svd(A)[2])
```

1.22.4 Write "type-stable" functions

When possible, it helps to ensure that a function always returns a value of the same type. Consider the following definition:

```
pos(x) = x < 0 ? 0 : x
```

Although this seems innocent enough, the problem is that `0` is an integer (of type `Int`) and `x` might be of any type. Thus, depending on the value of `x`, this function might return a value of either of two types. This behavior is allowed, and may be desirable in some cases. But it can easily be fixed as follows:

```
pos(x) = x < 0 ? zero(x) : x
```

There is also a `one` function, and a more general `oftype(x,y)` function, which returns `y` converted to the type of `x`. The first argument to any of these functions can be either a value or a type.

1.22.5 Avoid changing the type of a variable

An analogous "type-stability" problem exists for variables used repeatedly within a function:

```
function foo()
    x = 1
    for i = 1:10
        x = x/bar()
    end
    return x
end
```

Local variable `x` starts as an integer, and after one loop iteration becomes a floating-point number (the result of the `/` operator). This makes it more difficult for the compiler to optimize the body of the loop. There are several possible fixes:

- Initialize `x` with `x = 1.0`
- Declare the type of `x`: `x::Float64 = 1`
- Use an explicit conversion: `x = one(T)`

1.22.6 Separate kernel functions

Many functions follow a pattern of performing some set-up work, and then running many iterations to perform a core computation. Where possible, it is a good idea to put these core computations in separate functions. For example, the following contrived function returns an array of a randomly-chosen type:

```
function strange_twos(n)
    a = Array(randbool() ? Int64 : Float64, n)
    for i = 1:n
        a[i] = 2
    end
    return a
end
```

This should be written as:

```
function fill_twos!(a)
    for i=1:length(a)
        a[i] = 2
    end
end

function strange_twos(n)
    a = Array(randbool() ? Int64 : Float64, n)
    fill_twos!(a)
    return a
end
```

Julia's compiler specializes code for argument types at function boundaries, so in the original implementation it does not know the type of `a` during the loop (since it is chosen randomly). Therefore the second version is generally faster since the inner loop can be recompiled as part of `fill_twos!` for different types of `a`.

The second form is also often better style and can lead to more code reuse.

This pattern is used in several places in the standard library. For example, see `hvcat_fill` in abstractarray.jl, or the `fill!` function, which we could have used instead of writing our own `fill_twos!`.

Functions like `strange_twos` occur when dealing with data of uncertain type, for example data loaded from an input file that might contain either integers, floats, strings, or something else.

1.22.7 Tweaks

These are some minor points that might help in tight inner loops.

- Use `size(A,n)` when possible instead of `size(A)`.
- Avoid unnecessary arrays. For example, instead of `sum([x,y,z])` use `x+y+z`.

A Biblioteca Padrão de Julia (em inglês)

Release 0.2

Date 23/08/2014

2.1 Built-ins

2.1.1 Getting Around

exit (*[code]*)
Quit (or control-D at the prompt). The default exit code is zero, indicating that the processes completed successfully.

whos (*[Module,] [pattern::Regex]*)
Print information about global variables in a module, optionally restricted to those matching `pattern`.

edit (*file::String[, line]*)
Edit a file optionally providing a line number to edit at. Returns to the julia prompt when you quit the editor. If the file name ends in ".jl" it is reloaded when the editor closes the file.

edit (*function[, types]*)
Edit the definition of a function, optionally specifying a tuple of types to indicate which method to edit. When the editor exits, the source file containing the definition is reloaded.

require (*file::String...*)
Load source files once, in the context of the `Main` module, on every active node, searching the system-wide `LOAD_PATH` for files. `require` is considered a top-level operation, so it sets the current `include` path but does not use it to search for files (see help for `include`). This function is typically used to load library code, and is implicitly called by `using` to load packages.

reload (*file::String*)
Like `require`, except forces loading of files regardless of whether they have been loaded before. Typically used when interactively developing libraries.

include (*path::String*)
Evaluate the contents of a source file in the current context. During including, a task-local include path is set to the directory containing the file. Nested calls to `include` will search relative to that path. All paths refer to files on node 1 when running in parallel, and files will be fetched from node 1. This function is typically used to load source interactively, or to combine files in packages that are broken into multiple source files.

include_string(*code::String*)
Like include, except reads code from the given string rather than from a file. Since there is no file path involved, no path processing or fetching from node 1 is done.

evalfile(*path::String*)
Evaluate all expressions in the given file, and return the value of the last one. No other processing (path searching, fetching from node 1, etc.) is performed.

help(*name*)
Get help for a function. name can be an object or a string.

apropos(*string*)
Search documentation for functions related to string.

which(*f*, *args...*)
Show which method of f will be called for the given arguments.

methods(*f*)
Show all methods of f with their argument types.

methodswith(*typ*[, *showparents*])
Show all methods with an argument of type typ. If optional showparents is true, also show arguments with a parent type of typ, excluding type Any.

2.1.2 All Objects

is(*x*, *y*)
Determine whether x and y are identical, in the sense that no program could distinguish them.

isa(*x*, *type*)
Determine whether x is of the given type.

isequal(*x*, *y*)
True if and only if x and y have the same contents. Loosely speaking, this means x and y would look the same when printed.

isless(*x*, *y*)
Test whether x is less than y. Provides a total order consistent with isequal. Values that are normally unordered, such as NaN, are ordered in an arbitrary but consistent fashion. This is the default comparison used by sort. Non-numeric types that can be ordered should implement this function.

typeof(*x*)
Get the concrete type of x.

tuple(*xs...*)
Construct a tuple of the given objects.

ntuple(*n*, *f::Function*)
Create a tuple of length n, computing each element as f(i), where i is the index of the element.

object_id(*x*)
Get a unique integer id for x. object_id(x)==object_id(y) if and only if is(x,y).

hash(*x*)
Compute an integer hash code such that isequal(x,y) implies hash(x)==hash(y).

finalizer(*x*, *function*)
Register a function f(x) to be called when there are no program-accessible references to x. The behavior of this function is unpredictable if x is of a bits type.

copy(*x*)
Create a shallow copy of `x`: the outer structure is copied, but not all internal values. For example, copying an array produces a new array with identically-same elements as the original.

deepcopy(*x*)
Create a deep copy of `x`: everything is copied recursively, resulting in a fully independent object. For example, deep-copying an array produces a new array whose elements are deep-copies of the original elements.

As a special case, functions can only be actually deep-copied if they are anonymous, otherwise they are just copied. The difference is only relevant in the case of closures, i.e. functions which may contain hidden internal references.

While it isn't normally necessary, user-defined types can override the default `deepcopy` behavior by defining a specialized version of the function `deepcopy_internal(x::T, dict::ObjectIdDict)` (which shouldn't otherwise be used), where `T` is the type to be specialized for, and `dict` keeps track of objects copied so far within the recursion. Within the definition, `deepcopy_internal` should be used in place of `deepcopy`, and the `dict` variable should be updated as appropriate before returning.

convert(*type*, *x*)
Try to convert `x` to the given type.

promote(*xs...*)
Convert all arguments to their common promotion type (if any), and return them all (as a tuple).

2.1.3 Types

subtype(*type1*, *type2*)
True if and only if all values of `type1` are also of `type2`. Can also be written using the `<:` infix operator as `type1 <: type2`.

<:(*T1*, *T2*)
Subtype operator, equivalent to `subtype(T1,T2)`.

typemin(*type*)
The lowest value representable by the given (real) numeric type.

typemax(*type*)
The highest value representable by the given (real) numeric type.

realmin(*type*)
The smallest in absolute value non-denormal value representable by the given floating-point type

realmax(*type*)
The highest finite value representable by the given floating-point type

maxintfloat(*type*)
The largest integer losslessly representable by the given floating-point type

sizeof(*type*)
Size, in bytes, of the canonical binary representation of the given type, if any.

eps([*type*])
The distance between 1.0 and the next larger representable floating-point value of `type`. The only types that are sensible arguments are `Float32` and `Float64`. If `type` is omitted, then `eps(Float64)` is returned.

eps(*x*)
The distance between `x` and the next larger representable floating-point value of the same type as `x`.

promote_type(*type1*, *type2*)
Determine a type big enough to hold values of each argument type without loss, whenever possible. In some cases, where no type exists which to which both types can be promoted losslessly, some loss is tolerated; for

example, `promote_type(Int64,Float64)` returns `Float64` even though strictly, not all `Int64` values can be represented exactly as `Float64` values.

`getfield`(*value*, *name::Symbol*)
Extract a named field from a value of composite type. The syntax `a.b` calls `getfield(a, :b)`, and the syntax `a.(b)` calls `getfield(a, b)`.

`setfield`(*value*, *name::Symbol*, *x*)
Assign `x` to a named field in `value` of composite type. The syntax `a.b = c` calls `setfield(a, :b, c)`, and the syntax `a.(b) = c` calls `setfield(a, b, c)`.

`fieldtype`(*value*, *name::Symbol*)
Determine the declared type of a named field in a value of composite type.

2.1.4 Generic Functions

`method_exists`(*f*, *tuple*) → Bool
Determine whether the given generic function has a method matching the given tuple of argument types.

Example: `method_exists(length, (Array,)) = true`

`applicable`(*f*, *args...*)
Determine whether the given generic function has a method applicable to the given arguments.

`invoke`(*f*, *(types...)*, *args...*)
Invoke a method for the given generic function matching the specified types (as a tuple), on the specified arguments. The arguments must be compatible with the specified types. This allows invoking a method other than the most specific matching method, which is useful when the behavior of a more general definition is explicitly needed (often as part of the implementation of a more specific method of the same function).

`|`(*x*, *f*)
Applies a function to the preceding argument which allows for easy function chaining.

Example: `[1:5] | x->x.^2 | sum | inv`

2.1.5 Iteration

Sequential iteration is implemented by the methods `start`, `done`, and `next`. The general `for` loop:

```
for i = I
  # body
end
```

is translated to:

```
state = start(I)
while !done(I, state)
  (i, state) = next(I, state)
  # body
end
```

The `state` object may be anything, and should be chosen appropriately for each iterable type.

`start`(*iter*) → state
Get initial iteration state for an iterable object

`done`(*iter*, *state*) → Bool
Test whether we are done iterating

next (*iter*, *state*) → item, state
For a given iterable object and iteration state, return the current item and the next iteration state

zip (*iters...*)
For a set of iterable objects, returns an iterable of tuples, where the `i`th tuple contains the `i`th component of each input iterable.

Note that `zip` is it's own inverse: `[zip(zip(a...)...)...] == [a...]`.

enumerate (*iter*)
Return an iterator that yields `(i, x)` where `i` is an index starting at 1, and `x` is the `i`th value from the given iterator.

Fully implemented by: `Range`, `Range1`, `NDRange`, `Tuple`, `Real`, `AbstractArray`, `IntSet`, `ObjectIdDict`, `Dict`, `WeakKeyDict`, `EachLine`, `String`, `Set`, `Task`.

2.1.6 General Collections

isempty (*collection*) → Bool
Determine whether a collection is empty (has no elements).

empty! (*collection*) → collection
Remove all elements from a collection.

length (*collection*) → Integer
For ordered, indexable collections, the maximum index `i` for which `getindex(collection, i)` is valid. For unordered collections, the number of elements.

endof (*collection*) → Integer
Returns the last index of the collection.

Example: `endof([1,2,4]) = 3`

Fully implemented by: `Range`, `Range1`, `Tuple`, `Number`, `AbstractArray`, `IntSet`, `Dict`, `WeakKeyDict`, `String`, `Set`.

2.1.7 Iterable Collections

contains (*itr*, *x*) → Bool
Determine whether a collection contains the given value, `x`.

findin (*a*, *b*)
Returns the indices of elements in collection `a` that appear in collection `b`

unique (*itr*)
Returns an array containing only the unique elements of the iterable `itr`.

reduce (*op*, *v0*, *itr*)
Reduce the given collection with the given operator, i.e. accumulate `v = op(v,elt)` for each element, where `v` starts as `v0`. Reductions for certain commonly-used operators are available in a more convenient 1-argument form: `max(itr)`, `min(itr)`, `sum(itr)`, `prod(itr)`, `any(itr)`, `all(itr)`.

max (*itr*)
Returns the largest element in a collection

min (*itr*)
Returns the smallest element in a collection

indmax (*itr*) → Integer
Returns the index of the maximum element in a collection

indmin (*itr*) → Integer
Returns the index of the minimum element in a collection

findmax (*itr*) -> (*x*, *index*)
Returns the maximum element and its index

findmin (*itr*) -> (*x*, *index*)
Returns the minimum element and its index

sum (*itr*)
Returns the sum of all elements in a collection

prod (*itr*)
Returns the product of all elements of a collection

any (*itr*) → Bool
Test whether any elements of a boolean collection are true

all (*itr*) → Bool
Test whether all elements of a boolean collection are true

count (*itr*) → Integer
Count the number of boolean elements in `itr` which are true.

countp (*p*, *itr*) → Integer
Count the number of elements in `itr` for which predicate `p` is true.

any (*p*, *itr*) → Bool
Determine whether any element of `itr` satisfies the given predicate.

all (*p*, *itr*) → Bool
Determine whether all elements of `itr` satisfy the given predicate.

map (*f*, *c*) → collection
Transform collection `c` by applying `f` to each element.

Example: `map((x) -> x * 2, [1, 2, 3]) = [2, 4, 6]`

map! (*function*, *collection*)
In-place version of `map()`.

mapreduce (*f*, *op*, *itr*)
Applies function `f` to each element in `itr` and then reduces the result using the binary function `op`.

Example: `mapreduce(x->x^2, +, [1:3]) == 1 + 4 + 9 == 14`

first (*coll*)
Get the first element of an ordered collection.

last (*coll*)
Get the last element of an ordered collection.

2.1.8 Indexable Collections

getindex (*collection*, *key...*)
Retrieve the value(s) stored at the given key or index within a collection. The syntax `a[i,j,...]` is converted by the compiler to `getindex(a, i, j, ...)`.

setindex! (*collection*, *value*, *key...*)
Store the given value at the given key or index within a collection. The syntax `a[i,j,...] = x` is converted by the compiler to `setindex!(a, x, i, j, ...)`.

Fully implemented by: `Array`, `DArray`, `AbstractArray`, `SubArray`, `ObjectIdDict`, `Dict`, `WeakKeyDict`, `String`.

Partially implemented by: `Range`, `Range1`, `Tuple`.

2.1.9 Associative Collections

`Dict` is the standard associative collection. Its implementation uses the `hash(x)` as the hashing function for the key, and `isequal(x,y)` to determine equality. Define these two functions for custom types to override how they are stored in a hash table.

`ObjectIdDict` is a special hash table where the keys are always object identities. `WeakKeyDict` is a hash table implementation where the keys are weak references to objects, and thus may be garbage collected even when referenced in a hash table.

Dicts can be created using a literal syntax: `{"A"=>1, "B"=>2}`. Use of curly brackets will create a `Dict` of type `Dict{Any,Any}`. Use of square brackets will attempt to infer type information from the keys and values (i.e. `["A"=>1, "B"=>2]` creates a `Dict{ASCIIString, Int64}`). To explicitly specify types use the syntax: `(KeyType=>ValueType)[...]`. For example, `(ASCIIString=>Int32)["A"=>1, "B"=>2]`.

As with arrays, `Dicts` may be created with comprehensions. For example, `{i => f(i) for i = 1:10}`.

`Dict{K,V}``()`
Construct a hashtable with keys of type K and values of type V

`has`(*collection*, *key*)
Determine whether a collection has a mapping for a given key.

`get`(*collection*, *key*, *default*)
Return the value stored for the given key, or the given default value if no mapping for the key is present.

`getkey`(*collection*, *key*, *default*)
Return the key matching argument `key` if one exists in `collection`, otherwise return `default`.

`delete!`(*collection*, *key*)
Delete the mapping for the given key in a collection.

`keys`(*collection*)
Return an array of all keys in a collection.

`values`(*collection*)
Return an array of all values in a collection.

`collect`(*collection*)
Return an array of all items in a collection. For associative collections, returns (key, value) tuples.

`merge`(*collection*, *others...*)
Construct a merged collection from the given collections.

`merge!`(*collection*, *others...*)
Update collection with pairs from the other collections

`filter`(*function*, *collection*)
Return a copy of collection, removing (key, value) pairs for which function is false.

`filter!`(*function*, *collection*)
Update collection, removing (key, value) pairs for which function is false.

`eltype`(*collection*)
Returns the type tuple of the (key,value) pairs contained in collection.

sizehint (*s*, *n*)
Suggest that collection `s` reserve capacity for at least `n` elements. This can improve performance.

Fully implemented by: `ObjectIdDict`, `Dict`, `WeakKeyDict`.

Partially implemented by: `IntSet`, `Set`, `EnvHash`, `Array`.

2.1.10 Set-Like Collections

add! (*collection*, *key*)
Add an element to a set-like collection.

add_each! (*collection*, *iterable*)
Adds each element in iterable to the collection.

Set (*x...*)
Construct a `Set` with the given elements. Should be used instead of `IntSet` for sparse integer sets.

IntSet (*i...*)
Construct an `IntSet` of the given integers. Implemented as a bit string, and therefore good for dense integer sets.

union (*s1*, *s2...*)
Construct the union of two or more sets. Maintains order with arrays.

union! (*s1*, *s2*)
Constructs the union of IntSets s1 and s2, stores the result in `s1`.

intersect (*s1*, *s2...*)
Construct the intersection of two or more sets. Maintains order with arrays.

setdiff (*s1*, *s2*)
Construct the set of elements in `s1` but not `s2`. Maintains order with arrays.

symdiff (*s1*, *s2...*)
Construct the symmetric difference of elements in the passed in sets or arrays. Maintains order with arrays.

symdiff! (*s*, *n*)
IntSet s is destructively modified to toggle the inclusion of integer `n`.

symdiff! (*s*, *itr*)
For each element in `itr`, destructively toggle its inclusion in set `s`.

symdiff! (*s1*, *s2*)
Construct the symmetric difference of IntSets `s1` and `s2`, storing the result in `s1`.

complement (*s*)
Returns the set-complement of IntSet s.

complement! (*s*)
Mutates IntSet s into its set-complement.

del_each! (*s*, *itr*)
Deletes each element of itr in set s in-place.

intersect! (*s1*, *s2*)
Intersects IntSets s1 and s2 and overwrites the set s1 with the result. If needed, s1 will be expanded to the size of s2.

Fully implemented by: `IntSet`, `Set`.

Partially implemented by: `Array`.

2.1.11 Dequeues

push! (*collection*, *item*) → collection
Insert an item at the end of a collection.

pop! (*collection*) → item
Remove the last item in a collection and return it.

unshift! (*collection*, *item*) → collection
Insert an item at the beginning of a collection.

shift! (*collection*) → item
Remove the first item in a collection.

insert! (*collection*, *index*, *item*)
Insert an item at the given index.

delete! (*collection*, *index*) → item
Remove the item at the given index, and return the deleted item.

delete! (*collection*, *range*) → items
Remove items at specified range, and return a collection containing the deleted items.

resize! (*collection*, *n*) → collection
Resize collection to contain `n` elements.

append! (*collection*, *items*) → collection
Add the elements of `items` to the end of a collection.

Fully implemented by: `Vector` (aka 1-d `Array`).

2.1.12 Strings

length (*s*)
The number of characters in string `s`.

***** (*s*, *t*)
Concatenate strings.

Example: `"Hello "* "world"== "Hello world"`

^ (*s*, *n*)
Repeat string `s` `n` times.

Example: `"Julia "^3 == "Julia Julia Julia "`

string (*xs...*)
Create a string from any values using the `print` function.

repr (*x*)
Create a string from any value using the `show` function.

bytestring (*::Ptr{Uint8}*)
Create a string from the address of a C (0-terminated) string. A copy is made; the ptr can be safely freed.

bytestring (*s*)
Convert a string to a contiguous byte array representation appropriate for passing it to C functions.

ascii (*::Array{Uint8, 1}*)
Create an ASCII string from a byte array.

ascii (*s*)
Convert a string to a contiguous ASCII string (all characters must be valid ASCII characters).

utf8 (*::Array{Uint8, 1}*)
Create a UTF-8 string from a byte array.

utf8 (*s*)
Convert a string to a contiguous UTF-8 string (all characters must be valid UTF-8 characters).

is_valid_ascii (*s*) → Bool
Returns true if the string or byte vector is valid ASCII, false otherwise.

is_valid_utf8 (*s*) → Bool
Returns true if the string or byte vector is valid UTF-8, false otherwise.

is_valid_char (*c*) → Bool
Returns true if the given char or integer is a valid Unicode code point.

ismatch (*r::Regex*, *s::String*)
Test whether a string contains a match of the given regular expression.

lpad (*string*, *n*, *p*)
Make a string at least `n` characters long by padding on the left with copies of `p`.

rpad (*string*, *n*, *p*)
Make a string at least `n` characters long by padding on the right with copies of `p`.

search (*string*, *chars*[, *start*])
Search for the given characters within the given string. The second argument may be a single character, a vector or a set of characters, a string, or a regular expression (though regular expressions are only allowed on contiguous strings, such as ASCII or UTF-8 strings). The third argument optionally specifies a starting index. The return value is a range of indexes where the matching sequence is found, such that `s[search(s,x)] == x`. The return value is `0:-1` if there is no match.

replace (*string*, *pat*, *r*[, *n*])
Search for the given pattern `pat`, and replace each occurance with `r`. If `n` is provided, replace at most `n` occurances. As with search, the second argument may be a single character, a vector or a set of characters, a string, or a regular expression. If `r` is a function, each occurrence is replaced with `r(s)` where `s` is the matched substring.

split (*string, [chars, [limit,] [include_empty]]*)
Return an array of strings by splitting the given string on occurrences of the given character delimiters, which may be specified in any of the formats allowed by `search`'s second argument (i.e. a single character, collection of characters, string, or regular expression). If `chars` is omitted, it defaults to the set of all space characters, and `include_empty` is taken to be false. The last two arguments are also optional: they are are a maximum size for the result and a flag determining whether empty fields should be included in the result.

strip (*string*[, *chars*])
Return `string` with any leading and trailing whitespace removed. If a string `chars` is provided, instead remove characters contained in that string.

lstrip (*string*[, *chars*])
Return `string` with any leading whitespace removed. If a string `chars` is provided, instead remove characters contained in that string.

rstrip (*string*[, *chars*])
Return `string` with any trailing whitespace removed. If a string `chars` is provided, instead remove characters contained in that string.

beginswith (*string*, *prefix*)
Returns `true` if `string` starts with `prefix`.

endswith (*string*, *suffix*)
Returns `true` if `string` ends with `suffix`.

uppercase(*string*)
Returns `string` with all characters converted to uppercase.

lowercase(*string*)
Returns `string` with all characters converted to lowercase.

join(*strings*, *delim*)
Join an array of strings into a single string, inserting the given delimiter between adjacent strings.

chop(*string*)
Remove the last character from a string

chomp(*string*)
Remove a trailing newline from a string

ind2chr(*string*, *i*)
Convert a byte index to a character index

chr2ind(*string*, *i*)
Convert a character index to a byte index

isvalid(*str*, *i*)
Tells whether index `i` is valid for the given string

nextind(*str*, *i*)
Get the next valid string index after `i`. Returns `endof(str)+1` at the end of the string.

prevind(*str*, *i*)
Get the previous valid string index before `i`. Returns `0` at the beginning of the string.

thisind(*str*, *i*)
Adjust `i` downwards until it reaches a valid index for the given string.

randstring(*len*)
Create a random ASCII string of length `len`, consisting of upper- and lower-case letters and the digits 0-9

charwidth(*c*)
Gives the number of columns needed to print a character.

strwidth(*s*)
Gives the number of columns needed to print a string.

isalnum(*c::Char*)
Tests whether a character is alphanumeric.

isalpha(*c::Char*)
Tests whether a character is alphabetic.

isascii(*c::Char*)
Tests whether a character belongs to the ASCII character set.

isblank(*c::Char*)
Tests whether a character is a tab or space.

iscntrl(*c::Char*)
Tests whether a character is a control character.

isdigit(*c::Char*)
Tests whether a character is a numeric digit (0-9).

isgraph(*c::Char*)
Tests whether a character is printable, and not a space.

islower(*c::Char*)
Tests whether a character is a lowercase letter.

isprint(*c::Char*)
Tests whether a character is printable, including space.

ispunct(*c::Char*)
Tests whether a character is printable, and not a space or alphanumeric.

isspace(*c::Char*)
Tests whether a character is any whitespace character.

isupper(*c::Char*)
Tests whether a character is an uppercase letter.

isxdigit(*c::Char*)
Tests whether a character is a valid hexadecimal digit.

2.1.13 I/O

STDOUT
Global variable referring to the standard out stream.

STDERR
Global variable referring to the standard error stream.

STDIN
Global variable referring to the standard input stream.

OUTPUT_STREAM
The default stream used for text output, e.g. in the `print` and `show` functions.

open(*file_name*[, *read*, *write*, *create*, *truncate*, *append*]) → IOStream
Open a file in a mode specified by five boolean arguments. The default is to open files for reading only. Returns a stream for accessing the file.

open(*file_name*[, *mode*]) → IOStream
Alternate syntax for open, where a string-based mode specifier is used instead of the five booleans. The values of `mode` correspond to those from `fopen(3)` or Perl `open`, and are equivalent to setting the following boolean groups:

r	read
r+	read, write
w	write, create, truncate
w+	read, write, create, truncate
a	write, create, append
a+	read, write, create, append

open(*f::function*, *args...*)
Apply the function `f` to the result of `open(args...)` and close the resulting file descriptor upon completion.

Example: `open(readall, "file.txt")`

memio([*size*[, *finalize::Bool*]]) → IOStream
Create an in-memory I/O stream, optionally specifying how much initial space is needed.

fdio([*name::String*], *fd::Integer*[, *own::Bool*]) → IOStream
Create an `IOStream` object from an integer file descriptor. If `own` is true, closing this object will close the underlying descriptor. By default, an `IOStream` is closed when it is garbage collected. `name` allows you to associate the descriptor with a named file.

flush (*stream*)
Commit all currently buffered writes to the given stream.

close (*stream*)
Close an I/O stream. Performs a `flush` first.

write (*stream*, *x*)
Write the canonical binary representation of a value to the given stream.

read (*stream*, *type*)
Read a value of the given type from a stream, in canonical binary representation.

read (*stream*, *type*, *dims*)
Read a series of values of the given type from a stream, in canonical binary representation. `dims` is either a tuple or a series of integer arguments specifying the size of `Array` to return.

position (*s*)
Get the current position of a stream.

seek (*s*, *pos*)
Seek a stream to the given position.

seek_end (*s*)
Seek a stream to the end.

skip (*s*, *offset*)
Seek a stream relative to the current position.

eof (*stream*)
Tests whether an I/O stream is at end-of-file. If the stream is not yet exhausted, this function will block to wait for more data if necessary, and then return `false`. Therefore it is always safe to read one byte after seeing `eof` return `false`.

2.1.14 Text I/O

show (*x*)
Write an informative text representation of a value to the current output stream. New types should overload `show(io, x)` where the first argument is a stream.

print (*x*)
Write (to the default output stream) a canonical (un-decorated) text representation of a value if there is one, otherwise call `show`.

println (*x*)
Print (using `print()`) `x` followed by a newline

@printf ([*io::IOStream*], *"%Fmt"*, *args...*)
Print arg(s) using C `printf()` style format specification string. Optionally, an IOStream may be passed as the first argument to redirect output.

@sprintf (*"%Fmt"*, *args...*)
Return `@printf` formatted output as string.

showall (*x*)
Show x, printing all elements of arrays

dump (*x*)
Write a thorough text representation of a value to the current output stream.

readall (*stream*)
Read the entire contents of an I/O stream as a string.

readline (*stream*)
Read a single line of text, including a trailing newline character (if one is reached before the end of the input).

readuntil (*stream*, *delim*)
Read a string, up to and including the given delimiter byte.

readlines (*stream*)
Read all lines as an array.

eachline (*stream*)
Create an iterable object that will yield each line from a stream.

readdlm (*filename*, *delim::Char*)
Read a matrix from a text file where each line gives one row, with elements separated by the given delimeter. If all data is numeric, the result will be a numeric array. If some elements cannot be parsed as numbers, a cell array of numbers and strings is returned.

readdlm (*filename*, *delim::Char*, *T::Type*)
Read a matrix from a text file with a given element type. If `T` is a numeric type, the result is an array of that type, with any non-numeric elements as `NaN` for floating-point types, or zero. Other useful values of `T` include `ASCIIString`, `String`, and `Any`.

writedlm (*filename*, *array*, *delim::Char*)
Write an array to a text file using the given delimeter (defaults to comma).

readcsv (*filename*[, *T::Type*])
Equivalent to `readdlm` with `delim` set to comma.

writecsv (*filename*, *array*)
Equivalent to `writedlm` with `delim` set to comma.

2.1.15 Memory-mapped I/O

mmap_array (*type*, *dims*, *stream*[, *offset*])
Create an array whose values are linked to a file, using memory-mapping. This provides a convenient way of working with data too large to fit in the computer's memory.

The type determines how the bytes of the array are interpreted (no format conversions are possible), and dims is a tuple containing the size of the array.

The file is specified via the stream. When you initialize the stream, use "r" for a "read-only" array, and "w+" to create a new array used to write values to disk. Optionally, you can specify an offset (in bytes) if, for example, you want to skip over a header in the file.

Example: A = mmap_array(Int64, (25,30000), s)

This would create a 25-by-30000 array of Int64s, linked to the file associated with stream s.

msync (*array*)
Forces synchronization between the in-memory version of a memory-mapped array and the on-disk version. You may not need to call this function, because synchronization is performed at intervals automatically by the operating system. Hower, you can call this directly if, for example, you are concerned about losing the result of a long-running calculation.

mmap (*len*, *prot*, *flags*, *fd*, *offset*)
Low-level interface to the mmap system call. See the man page.

munmap (*pointer*, *len*)
Low-level interface for unmapping memory (see the man page). With mmap_array you do not need to call this directly; the memory is unmapped for you when the array goes out of scope.

2.1.16 Standard Numeric Types

`Bool Int8 Uint8 Int16 Uint16 Int32 Uint32 Int64 Uint64 Float32 Float64 Complex64 Complex128`

2.1.17 Mathematical Functions

- (*x*)
Unary minus operator.

+ (*x*, *y*)
Binary addition operator.

- (*x*, *y*)
Binary subtraction operator.

***** (*x*, *y*)
Binary multiplication operator.

/ (*x*, *y*)
Binary left-division operator.

**** (*x*, *y*)
Binary right-division operator.

^ (*x*, *y*)
Binary exponentiation operator.

.+ (*x*, *y*)
Element-wise binary addition operator.

.- (*x*, *y*)
Element-wise binary subtraction operator.

.* (*x*, *y*)
Element-wise binary multiplication operator.

./ (*x*, *y*)
Element-wise binary left division operator.

. (*x*, *y*)
Element-wise binary right division operator.

.^ (*x*, *y*)
Element-wise binary exponentiation operator.

div (*a*, *b*)
Compute a/b, truncating to an integer

fld (*a*, *b*)
Largest integer less than or equal to a/b

mod (*x*, *m*)
Modulus after division, returning in the range [0,m)

rem (*x*, *m*)
Remainder after division

% (*x*, *m*)
Remainder after division. The operator form of `rem`.

mod1 (*x*, *m*)
Modulus after division, returning in the range (0,m]

// (*num*, *den*)
Rational division

num (*x*)
Numerator of the rational representation of `x`

den (*x*)
Denominator of the rational representation of `x`

<< (*x*, *n*)
Left shift operator.

>> (*x*, *n*)
Right shift operator.

>>> (*x*, *n*)
Unsigned right shift operator.

: (*start*[, *step*], *stop*)
Range operator. `a:b` constructs a range from `a` to `b` with a step size of 1, and `a:s:b` is similar but uses a step size of `s`. These syntaxes call the function `colon`. The colon is also used in indexing to select whole dimensions.

colon (*start*[, *step*], *stop*)
Called by `:` syntax for constructing ranges.

== (*x*, *y*)
Equality comparison operator.

!= (*x*, *y*)
Not-equals comparison operator.

< (*x*, *y*)
Less-than comparison operator.

<= (*x*, *y*)
Less-than-or-equals comparison operator.

> (*x*, *y*)
Greater-than comparison operator.

>= (*x*, *y*)
Greater-than-or-equals comparison operator.

.== (*x*, *y*)
Element-wise equality comparison operator.

.!= (*x*, *y*)
Element-wise not-equals comparison operator.

.< (*x*, *y*)
Element-wise less-than comparison operator.

.<= (*x*, *y*)
Element-wise less-than-or-equals comparison operator.

.> (*x*, *y*)
Element-wise greater-than comparison operator.

.>= (*x*, *y*)
Element-wise greater-than-or-equals comparison operator.

cmp (*x*, *y*)
Return -1, 0, or 1 depending on whether `x<y`, `x==y`, or `x>y`, respectively

! (*x*)
Boolean not

~ (*x*)
Bitwise not

& (*x*, *y*)
Bitwise and

| (*x*, *y*)
Bitwise or

$ (*x*, *y*)
Bitwise exclusive or

sin (*x*)
Compute sine of `x`, where `x` is in radians

cos (*x*)
Compute cosine of `x`, where `x` is in radians

tan (*x*)
Compute tangent of `x`, where `x` is in radians

sind (*x*)
Compute sine of `x`, where `x` is in degrees

cosd (*x*)
Compute cosine of `x`, where `x` is in degrees

tand (*x*)
Compute tangent of `x`, where `x` is in degrees

sinh (*x*)
Compute hyperbolic sine of `x`

cosh (*x*)
Compute hyperbolic cosine of `x`

tanh (*x*)
Compute hyperbolic tangent of `x`

asin (*x*)
Compute the inverse sine of `x`, where the output is in radians

acos (*x*)
Compute the inverse cosine of `x`, where the output is in radians

atan (*x*)
Compute the inverse tangent of `x`, where the output is in radians

atan2 (*y*, *x*)
Compute the inverse tangent of `y/x`, using the signs of both `x` and `y` to determine the quadrant of the return value.

asind (*x*)
Compute the inverse sine of `x`, where the output is in degrees

acosd (*x*)
Compute the inverse cosine of `x`, where the output is in degrees

atand(*x*)
Compute the inverse tangent of x, where the output is in degrees

sec(*x*)
Compute the secant of x, where x is in radians

csc(*x*)
Compute the cosecant of x, where x is in radians

cot(*x*)
Compute the cotangent of x, where x is in radians

secd(*x*)
Compute the secant of x, where x is in degrees

cscd(*x*)
Compute the cosecant of x, where x is in degrees

cotd(*x*)
Compute the cotangent of x, where x is in degrees

asec(*x*)
Compute the inverse secant of x, where the output is in radians

acsc(*x*)
Compute the inverse cosecant of x, where the output is in radians

acot(*x*)
Compute the inverse cotangent of x, where the output is in radians

asecd(*x*)
Compute the inverse secant of x, where the output is in degrees

acscd(*x*)
Compute the inverse cosecant of x, where the output is in degrees

acotd(*x*)
Compute the inverse cotangent of x, where the output is in degrees

sech(*x*)
Compute the hyperbolic secant of x

csch(*x*)
Compute the hyperbolic cosecant of x

coth(*x*)
Compute the hyperbolic cotangent of x

asinh(*x*)
Compute the inverse hyperbolic sine of x

acosh(*x*)
Compute the inverse hyperbolic cosine of x

atanh(*x*)
Compute the inverse hyperbolic cotangent of x

asech(*x*)
Compute the inverse hyperbolic secant of x

acsch(*x*)
Compute the inverse hyperbolic cosecant of x

acoth (*x*)
Compute the inverse hyperbolic cotangent of `x`

sinc (*x*)
Compute $\sin(\pi x)/(\pi x)$ if $x \neq 0$, and 1 if $x = 0$.

cosc (*x*)
Compute $\cos(\pi x)/x - \sin(\pi x)/(\pi x^2)$ if $x \neq 0$, and 0 if $x = 0$. This is the derivative of `sinc(x)`.

degrees2radians (*x*)
Convert `x` from degrees to radians

radians2degrees (*x*)
Convert `x` from radians to degrees

hypot (*x*, *y*)
Compute the $\sqrt{x^2 + y^2}$ without undue overflow or underflow

log (*x*)
Compute the natural logarithm of `x`

log2 (*x*)
Compute the natural logarithm of `x` to base 2

log10 (*x*)
Compute the natural logarithm of `x` to base 10

log1p (*x*)
Accurate natural logarithm of `1+x`

frexp (*val*, *exp*)
Return a number `x` such that it has a magnitude in the interval `[1/2, 1)` or 0, and val = $x \times 2^{exp}$.

exp (*x*)
Compute e^x

exp2 (*x*)
Compute 2^x

ldexp (*x*, *n*)
Compute $x \times 2^n$

modf (*x*)
Return a tuple (fpart,ipart) of the fractional and integral parts of a number. Both parts have the same sign as the argument.

expm1 (*x*)
Accurately compute $e^x - 1$

square (*x*)
Compute x^2

round (*x*[, *digits*[, *base*]]) → FloatingPoint
`round(x)` returns the nearest integer to `x`. `round(x, digits)` rounds to the specified number of digits after the decimal place, or before if negative, e.g., `round(pi,2)` is `3.14`. `round(x, digits, base)` rounds using a different base, defaulting to 10, e.g., `round(pi, 3, 2)` is `3.125`.

ceil (*x*[, *digits*[, *base*]]) → FloatingPoint
Returns the nearest integer not less than `x`. `digits` and `base` work as above.

floor (*x*[, *digits*[, *base*]]) → FloatingPoint
Returns the nearest integer not greater than `x`. `digits` and `base` work as above.

trunc (*x*[, *digits*[, *base*]]) → FloatingPoint
Returns the nearest integer not greater in magnitude than x. digits and base work as above.

iround (*x*) → Integer
Returns the nearest integer to x.

iceil (*x*) → Integer
Returns the nearest integer not less than x.

ifloor (*x*) → Integer
Returns the nearest integer not greater than x.

itrunc (*x*) → Integer
Returns the nearest integer not greater in magnitude than x.

signif (*x*, *digits*[, *base*]) → FloatingPoint
Rounds (in the sense of round) x so that there are digits significant digits, under a base base representation, default 10. E.g., signif(123.456, 2) is 120.0, and signif(357.913, 4, 2) is 352.0.

min (*x*, *y*)
Return the minimum of x and y

max (*x*, *y*)
Return the maximum of x and y

clamp (*x*, *lo*, *hi*)
Return x if lo <= x <= y. If x < lo, return lo. If x > hi, return hi.

abs (*x*)
Absolute value of x

abs2 (*x*)
Squared absolute value of x

copysign (*x*, *y*)
Return x such that it has the same sign as y

sign (*x*)
Return +1 if x is positive, 0 if x == 0, and −1 if x is negative.

signbit (*x*)
Returns 1 if the value of the sign of x is negative, otherwise 0.

flipsign (*x*, *y*)
Return x with its sign flipped if y is negative. For example abs(x) = flipsign(x,x).

sqrt (*x*)
Return $\sqrt{x}$

cbrt (*x*)
Return $x^{1/3}$

erf (*x*)
Compute the error function of x, defined by $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$ for arbitrary complex x.

erfc (*x*)
Compute the complementary error function of x, defined by $1 - \operatorname{erf}(x)$.

erfcx (*x*)
Compute the scaled complementary error function of x, defined by $e^{x^2} \operatorname{erfc}(x)$. Note also that $\operatorname{erfcx}(-ix)$ computes the Faddeeva function $w(x)$.

erfi (*x*)
Compute the imaginary error function of x, defined by $-i \operatorname{erf}(ix)$.

dawson(*x*)
Compute the Dawson function (scaled imaginary error function) of `x`, defined by $\frac{\sqrt{\pi}}{2}e^{-x^2}\,\mathrm{erfi}(x)$.

real(*z*)
Return the real part of the complex number `z`

imag(*z*)
Return the imaginary part of the complex number `z`

reim(*z*)
Return both the real and imaginary parts of the complex number `z`

conj(*z*)
Compute the complex conjugate of a complex number `z`

angle(*z*)
Compute the phase angle of a complex number `z`

cis(*z*)
Return `cos(z) + i*sin(z)` if `z` is real. Return `(cos(real(z)) + i*sin(real(z)))/exp(imag(z))` if `z` is complex

binomial(*n*, *k*)
Number of ways to choose `k` out of `n` items

factorial(*n*)
Factorial of n

factorial(*n*, *k*)
Compute `factorial(n)/factorial(k)`

factor(*n*)
Compute the prime factorization of an integer `n`. Returns a dictionary. The keys of the dictionary correspond to the factors, and hence are of the same type as `n`. The value associated with each key indicates the number of times the factor appears in the factorization.

Example: $100 = 2 * 2 * 5 * 5$; then, `factor(100) -> [5=>2,2=>2]`

gcd(*x*, *y*)
Greatest common divisor

lcm(*x*, *y*)
Least common multiple

gcdx(*x*, *y*)
Greatest common divisor, also returning integer coefficients `u` and `v` that solve `ux+vy == gcd(x,y)`

ispow2(*n*)
Test whether `n` is a power of two

nextpow2(*n*)
Next power of two not less than `n`

prevpow2(*n*)
Previous power of two not greater than `n`

nextpow(*a*, *n*)
Next power of `a` not less than `n`

prevpow(*a*, *n*)
Previous power of `a` not greater than `n`

nextprod($[a, b, c]$, *n*)
Next integer not less than `n` that can be written `a^i1 * b^i2 * c^i3` for integers `i1`, `i2`, `i3`.

prevprod([*a*, *b*, *c*], *n*)
Previous integer not greater than `n` that can be written `a^i1 * b^i2 * c^i3` for integers `i1`, `i2`, `i3`.

invmod(*x*, *m*)
Inverse of `x`, modulo `m`

powermod(*x*, *p*, *m*)
Compute `mod(x^p, m)`

gamma(*x*)
Compute the gamma function of `x`

lgamma(*x*)
Compute the logarithm of `gamma(x)`

lfact(*x*)
Compute the logarithmic factorial of `x`

digamma(*x*)
Compute the digamma function of `x` (the logarithmic derivative of `gamma(x)`)

airy(*k*, *x*)
kth derivative of the Airy function $\mathrm{Ai}(x)$.

airyai(*x*)
Airy function $\mathrm{Ai}(x)$.

airyprime(*x*)
Airy function derivative $\mathrm{Ai}'(x)$.

airyaiprime(*x*)
Airy function derivative $\mathrm{Ai}'(x)$.

airybi(*x*)
Airy function $\mathrm{Bi}(x)$.

airybiprime(*x*)
Airy function derivative $\mathrm{Bi}'(x)$.

besselj0(*x*)
Bessel function of the first kind of order 0, $J_0(x)$.

besselj1(*x*)
Bessel function of the first kind of order 1, $J_1(x)$.

besselj(*nu*, *x*)
Bessel function of the first kind of order `nu`, $J_\nu(x)$.

bessely0(*x*)
Bessel function of the second kind of order 0, $Y_0(x)$.

bessely1(*x*)
Bessel function of the second kind of order 1, $Y_1(x)$.

bessely(*nu*, *x*)
Bessel function of the second kind of order `nu`, $Y_\nu(x)$.

hankelh1(*nu*, *x*)
Bessel function of the third kind of order `nu`, $H_\nu^{(1)}(x)$.

hankelh2(*nu*, *x*)
Bessel function of the third kind of order `nu`, $H_\nu^{(2)}(x)$.

besseli (*nu*, *x*)
Modified Bessel function of the first kind of order `nu`, $I_\nu(x)$.

besselk (*nu*, *x*)
Modified Bessel function of the second kind of order `nu`, $K_\nu(x)$.

beta (*x*, *y*)
Euler integral of the first kind $\mathrm{B}(x,y) = \Gamma(x)\Gamma(y)/\Gamma(x+y)$.

lbeta (*x*, *y*)
Natural logarithm of the beta function $\log(\mathrm{B}(x,y))$.

eta (*x*)
Dirichlet eta function $\eta(s) = \sum_{n=1}^{\infty}(-)^{n-1}/n^s$.

zeta (*x*)
Riemann zeta function $\zeta(s)$.

bitmix (*x*, *y*)
Hash two integers into a single integer. Useful for constructing hash functions.

ndigits (*n*, *b*)
Compute the number of digits in number `n` written in base `b`.

2.1.18 Data Formats

bin (*n*[, *pad*])
Convert an integer to a binary string, optionally specifying a number of digits to pad to.

hex (*n*[, *pad*])
Convert an integer to a hexadecimal string, optionally specifying a number of digits to pad to.

dec (*n*[, *pad*])
Convert an integer to a decimal string, optionally specifying a number of digits to pad to.

oct (*n*[, *pad*])
Convert an integer to an octal string, optionally specifying a number of digits to pad to.

base (*base*, *n*[, *pad*])
Convert an integer to a string in the given base, optionally specifying a number of digits to pad to. The base can be specified as either an integer, or as a `Uint8` array of character values to use as digit symbols.

bits (*n*)
A string giving the literal bit representation of a number.

parseint ([*type*], *str*[, *base*])
Parse a string as an integer in the given base (default 10), yielding a number of the specified type (default `Int`).

parsefloat ([*type*], *str*)
Parse a string as a decimal floating point number, yielding a number of the specified type.

bool (*x*)
Convert a number or numeric array to boolean

isbool (*x*)
Test whether number or array is boolean

int (*x*)
Convert a number or array to the default integer type on your platform. Alternatively, `x` can be a string, which is parsed as an integer.

uint (*x*)
Convert a number or array to the default unsigned integer type on your platform. Alternatively, `x` can be a string, which is parsed as an unsigned integer.

integer (*x*)
Convert a number or array to integer type. If `x` is already of integer type it is unchanged, otherwise it converts it to the default integer type on your platform.

isinteger (*x*)
Test whether a number or array is of integer type

signed (*x*)
Convert a number to a signed integer

unsigned (*x*)
Convert a number to an unsigned integer

int8 (*x*)
Convert a number or array to `Int8` data type

int16 (*x*)
Convert a number or array to `Int16` data type

int32 (*x*)
Convert a number or array to `Int32` data type

int64 (*x*)
Convert a number or array to `Int64` data type

int128 (*x*)
Convert a number or array to `Int128` data type

uint8 (*x*)
Convert a number or array to `Uint8` data type

uint16 (*x*)
Convert a number or array to `Uint16` data type

uint32 (*x*)
Convert a number or array to `Uint32` data type

uint64 (*x*)
Convert a number or array to `Uint64` data type

uint128 (*x*)
Convert a number or array to `Uint128` data type

float32 (*x*)
Convert a number or array to `Float32` data type

float64 (*x*)
Convert a number or array to `Float64` data type

float (*x*)
Convert a number, array, or string to a `FloatingPoint` data type. For numeric data, the smallest suitable `FloatingPoint` type is used. For strings, it converts to `Float64`.

significand (*x*)
Extract the significand(s) (a.k.a. mantissa), in binary representation, of a floating-point number or array.

For example, `significand(15.2)/15.2 == 0.125`, and `significand(15.2)*8 == 15.2`

exponent (*x*) → Int
Get the exponent of a normalized floating-point number.

float64_valued(*x::Rational*)
True if `x` can be losslessly represented as a `Float64` data type

complex64(*r*, *i*)
Convert to `r+i*im` represented as a `Complex64` data type

complex128(*r*, *i*)
Convert to `r+i*im` represented as a `Complex128` data type

char(*x*)
Convert a number or array to `Char` data type

complex(*r*, *i*)
Convert real numbers or arrays to complex

iscomplex(*x*) → Bool
Test whether a number or array is of a complex type

isreal(*x*) → Bool
Test whether a number or array is of a real type

bswap(*n*)
Byte-swap an integer

num2hex(*f*)
Get a hexadecimal string of the binary representation of a floating point number

hex2num(*str*)
Convert a hexadecimal string to the floating point number it represents

2.1.19 Numbers

one(*x*)
Get the multiplicative identity element for the type of x (x can also specify the type itself). For matrices, returns an identity matrix of the appropriate size and type.

zero(*x*)
Get the additive identity element for the type of x (x can also specify the type itself).

pi
The constant pi

im
The imaginary unit

e
The constant e

Inf
Positive infinity of type Float64

Inf32
Positive infinity of type Float32

NaN
A not-a-number value of type Float64

NaN32
A not-a-number value of type Float32

isdenormal(*f*) → Bool
Test whether a floating point number is denormal

isfinite(*f*) → Bool
Test whether a number is finite

isinf(*f*)
Test whether a number is infinite

isnan(*f*)
Test whether a floating point number is not a number (NaN)

inf(*f*)
Returns infinity in the same floating point type as `f` (or `f` can by the type itself)

nan(*f*)
Returns NaN in the same floating point type as `f` (or `f` can by the type itself)

nextfloat(*f*)
Get the next floating point number in lexicographic order

prevfloat(*f*) → Float
Get the previous floating point number in lexicographic order

integer_valued(*x*)
Test whether `x` is numerically equal to some integer

real_valued(*x*)
Test whether `x` is numerically equal to some real number

BigInt(*x*)
Create an arbitrary precision integer. `x` may be an `Int` (or anything that can be converted to an `Int`) or a `String`. The usual mathematical operators are defined for this type, and results are promoted to a `BigInt`.

BigFloat(*x*)
Create an arbitrary precision floating point number. `x` may be an `Integer`, a `Float64`, a `String` or a `BigInt`. The usual mathematical operators are defined for this type, and results are promoted to a `BigFloat`.

Integers

count_ones(*x::Integer*) → Integer
Number of ones in the binary representation of `x`.

Example: `count_ones(7) -> 3`

count_zeros(*x::Integer*) → Integer
Number of zeros in the binary representation of `x`.

Example: `count_zeros(int32(2 ^ 16 - 1)) -> 16`

leading_zeros(*x::Integer*) → Integer
Number of zeros leading the binary representation of `x`.

Example: `leading_zeros(int32(1)) -> 31`

leading_ones(*x::Integer*) → Integer
Number of ones leading the binary representation of `x`.

Example: `leading_ones(int32(2 ^ 32 - 2)) -> 31`

trailing_zeros(*x::Integer*) → Integer
Number of zeros trailing the binary representation of `x`.

Example: `trailing_zeros(2) -> 1`

trailing_ones (*x::Integer*) → Integer
Number of ones trailing the binary representation of `x`.

Example: `trailing_ones(3) -> 2`

isprime (*x::Integer*) → Bool
Returns `true` if `x` is prime, and `false` otherwise.

Example: `isprime(3) -> true`

isodd (*x::Integer*) → Bool
Returns `true` if `x` is odd (that is, not divisible by 2), and `false` otherwise.

Example: `isodd(9) -> false`

iseven (*x::Integer*) → Bool
Returns `true` is `x` is even (that is, divisible by 2), and `false` otherwise.

Example: `iseven(1) -> false`

2.1.20 Random Numbers

Random number generateion in Julia uses the Mersenne Twister library. Julia has a global RNG, which is used by default. Multiple RNGs can be plugged in using the `AbstractRNG` object, which can then be used to have multiple streams of random numbers. Currently, only `MersenneTwister` is supported.

srand ([*rng*], *seed*)
Seed the RNG with a `seed`, which may be an unsigned integer or a vector of unsigned integers. `seed` can even be a filename, in which case the seed is read from a file. If the argument `rng` is not provided, the default global RNG is seeded.

MersenneTwister ([*seed*])
Create a `MersenneTwister` RNG object. Different RNG objects can have their own seeds, which may be useful for generating different streams of random numbers.

rand ()
Generate a `Float64` random number uniformly in [0,1)

rand! ([*rng*], *A*)
Populate the array A with random number generated from the specified RNG.

rand (*rng::AbstractRNG*[, *dims...*])
Generate a random `Float64` number or array of the size specified by dims, using the specified RNG object. Currently, `MersenneTwister` is the only available Random Number Generator (RNG), which may be seeded using srand.

rand (*dims or [dims...]*)
Generate a random `Float64` array of the size specified by dims

rand (*Int32|Uint32|Int64|Uint64|Int128|Uint128*[, *dims...*])
Generate a random integer of the given type. Optionally, generate an array of random integers of the given type by specifying dims.

rand (*r*[, *dims...*])
Generate a random integer from the inclusive interval specified by `Range1 r` (for example, `1:n`). Optionally, generate a random integer array.

randbool ([*dims...*])
Generate a random boolean value. Optionally, generate an array of random boolean values.

randbool! (*A*)
Fill an array with random boolean values. A may be an `Array` or a `BitArray`.

randn (*dims or [dims...]*)
Generate a normally-distributed random number with mean 0 and standard deviation 1. Optionally generate an array of normally-distributed random numbers.

2.1.21 Arrays

Basic functions

ndims (*A*) → Integer
Returns the number of dimensions of A

size (*A*)
Returns a tuple containing the dimensions of A

eltype (*A*)
Returns the type of the elements contained in A

length (*A*) → Integer
Returns the number of elements in A (note that this differs from MATLAB where `length(A)` is the largest dimension of `A`)

nnz (*A*)
Counts the number of nonzero values in array A (dense or sparse)

scale! (*A*, *k*)
Scale the contents of an array A with k (in-place)

conj! (*A*)
Convert an array to its complex conjugate in-place

stride (*A*, *k*)
Returns the distance in memory (in number of elements) between adjacent elements in dimension k

strides (*A*)
Returns a tuple of the memory strides in each dimension

Constructors

Array (*type*, *dims*)
Construct an uninitialized dense array. `dims` may be a tuple or a series of integer arguments.

getindex (*type*[, *elements...*])
Construct a 1-d array of the specified type. This is usually called with the syntax `Type[]`. Element values can be specified using `Type[a,b,c,...]`.

cell (*dims*)
Construct an uninitialized cell array (heterogeneous array). `dims` can be either a tuple or a series of integer arguments.

zeros (*type*, *dims*)
Create an array of all zeros of specified type

ones (*type*, *dims*)
Create an array of all ones of specified type

trues (*dims*)
Create a Bool array with all values set to true

falses (*dims*)
Create a Bool array with all values set to false

fill (*v*, *dims*)
Create an array filled with `v`

fill! (*A*, *x*)
Fill array `A` with value `x`

reshape (*A*, *dims*)
Create an array with the same data as the given array, but with different dimensions. An implementation for a particular type of array may choose whether the data is copied or shared.

similar (*array*, *element_type*, *dims*)
Create an uninitialized array of the same type as the given array, but with the specified element type and dimensions. The second and third arguments are both optional. The `dims` argument may be a tuple or a series of integer arguments.

reinterpret (*type*, *A*)
Construct an array with the same binary data as the given array, but with the specified element type

eye (*n*)
n-by-n identity matrix

eye (*m*, *n*)
m-by-n identity matrix

linspace (*start*, *stop*, *n*)
Construct a vector of `n` linearly-spaced elements from `start` to `stop`.

logspace (*start*, *stop*, *n*)
Construct a vector of `n` logarithmically-spaced numbers from `10^start` to `10^stop`.

Mathematical operators and functions

All mathematical operations and functions are supported for arrays

bsxfun (*fn*, *A*, *B*[, *C*...])
Apply binary function `fn` to two or more arrays, with singleton dimensions expanded.

Indexing, Assignment, and Concatenation

getindex (*A*, *ind*)
Returns a subset of array `A` as specified by `ind`, which may be an `Int`, a `Range`, or a `Vector`.

sub (*A*, *ind*)
Returns a SubArray, which stores the input `A` and `ind` rather than computing the result immediately. Calling `getindex` on a SubArray computes the indices on the fly.

slicedim (*A*, *d*, *i*)
Return all the data of `A` where the index for dimension `d` equals `i`. Equivalent to `A[:,:,...,i,:,:,...]` where `i` is in position `d`.

setindex! (*A*, *X*, *ind*)
Store values from array `X` within some subset of `A` as specified by `ind`.

cat (*dim*, *A*...)
Concatenate the input arrays along the specified dimension

vcat (*A...*)
Concatenate along dimension 1

hcat (*A...*)
Concatenate along dimension 2

hvcat (*rows::(Int...)*, *values...*)
Horizontal and vertical concatenation in one call. This function is called for block matrix syntax. The first argument specifies the number of arguments to concatenate in each block row. For example, `[a b;c d e]` calls `hvcat((2,3),a,b,c,d,e)`.

flipdim (*A*, *d*)
Reverse `A` in dimension `d`.

flipud (*A*)
Equivalent to `flipdim(A,1)`.

fliplr (*A*)
Equivalent to `flipdim(A,2)`.

circshift (*A*, *shifts*)
Circularly shift the data in an array. The second argument is a vector giving the amount to shift in each dimension.

find (*A*)
Return a vector of the linear indexes of the non-zeros in `A`.

findn (*A*)
Return a vector of indexes for each dimension giving the locations of the non-zeros in `A`.

nonzeros (*A*)
Return a vector of the non-zero values in array `A`.

findfirst (*A*)
Return the index of the first non-zero value in `A`.

findfirst (*A*, *v*)
Return the index of the first element equal to `v` in `A`.

findfirst (*predicate*, *A*)
Return the index of the first element that satisfies the given predicate in `A`.

permutedims (*A*, *perm*)
Permute the dimensions of array `A`. `perm` is a vector specifying a permutation of length `ndims(A)`. This is a generalization of transpose for multi-dimensional arrays. Transpose is equivalent to `permute(A,[2,1])`.

ipermutedims (*A*, *perm*)
Like `permutedims()`, except the inverse of the given permutation is applied.

squeeze (*A*, *dims*)
Remove the dimensions specified by `dims` from array `A`

vec (*Array*) → Vector
Vectorize an array using column-major convention.

Array functions

cumprod (*A*[, *dim*])
Cumulative product along a dimension.

cumsum (*A*[, *dim*])
Cumulative sum along a dimension.

cumsum_kbn (*A*[, *dim*])
Cumulative sum along a dimension, using the Kahan-Babuska-Neumaier compensated summation algorithm for additional accuracy.

cummin (*A*[, *dim*])
Cumulative minimum along a dimension.

cummax (*A*[, *dim*])
Cumulative maximum along a dimension.

diff (*A*[, *dim*])
Finite difference operator of matrix or vector.

rot180 (*A*)
Rotate matrix `A` 180 degrees.

rotl90 (*A*)
Rotate matrix `A` left 90 degrees.

rotr90 (*A*)
Rotate matrix `A` right 90 degrees.

reducedim (*f*, *A*, *dims*, *initial*)
Reduce 2-argument function `f` along dimensions of `A`. `dims` is a vector specifying the dimensions to reduce, and `initial` is the initial value to use in the reductions.

mapslices (*f*, *A*, *dims*)
Transform the given dimensions of array `A` using function `f`. `f` is called on each slice of `A` of the form `A[...,:,...,:,...]`. `dims` is an integer vector specifying where the colons go in this expression. The results are concatenated along the remaining dimensions. For example, if `dims` is `[1,2]` and A is 4-dimensional, `f` is called on `A[:,:,i,j]` for all `i` and `j`.

sum_kbn (*A*)
Returns the sum of all array elements, using the Kahan-Babuska-Neumaier compensated summation algorithm for additional accuracy.

2.1.22 Combinatorics

nthperm (*v*, *k*)
Compute the kth lexicographic permutation of a vector.

nthperm! (*v*, *k*)
In-place version of `nthperm()`.

randperm (*n*)
Construct a random permutation of the given length.

invperm (*v*)
Return the inverse permutation of v.

isperm (*v*) → Bool
Returns true if v is a valid permutation.

permute! (*v*, *p*)
Permute vector `v` in-place, according to permutation `p`. No checking is done to verify that `p` is a permutation.

To return a new permutation, use `v[p]`. Note that this is generally faster than `permute!(v,p)` for large vectors.

ipermute! (*v*, *p*)
Like permute!, but the inverse of the given permutation is applied.

randcycle(*n*)
Construct a random cyclic permutation of the given length.

shuffle(*v*)
Randomly rearrange the elements of a vector.

shuffle!(*v*)
In-place version of `shuffle()`.

reverse(*v*)
Reverse vector `v`.

reverse!(*v*) → v
In-place version of `reverse()`.

combinations(*array*, *n*)
Generate all combinations of `n` elements from a given array. Because the number of combinations can be very large, this function runs inside a Task to produce values on demand. Write `c = @task combinations(a,n)`, then iterate `c` or call `consume` on it.

integer_partitions(*n*, *m*)
Generate all arrays of `m` integers that sum to `n`. Because the number of partitions can be very large, this function runs inside a Task to produce values on demand. Write `c = @task integer_partitions(n,m)`, then iterate `c` or call `consume` on it.

partitions(*array*)
Generate all set partitions of the elements of an array, represented as arrays of arrays. Because the number of partitions can be very large, this function runs inside a Task to produce values on demand. Write `c = @task partitions(a)`, then iterate `c` or call `consume` on it.

2.1.23 Statistics

mean(*v*[, *region*])
Compute the mean of whole array `v`, or optionally along the dimensions in `region`.

std(*v*[, *region*])
Compute the sample standard deviation of a vector or array``v``, optionally along dimensions in `region`. The algorithm returns an estimator of the generative distribution's standard deviation under the assumption that each entry of `v` is an IID draw from that generative distribution. This computation is equivalent to calculating `sqrt(sum((v - mean(v)).^2) / (length(v) - 1))`.

stdm(*v*, *m*)
Compute the sample standard deviation of a vector `v` with known mean `m`.

var(*v*[, *region*])
Compute the sample variance of a vector or array``v``, optionally along dimensions in `region`. The algorithm will return an estimator of the generative distribution's variance under the assumption that each entry of `v` is an IID draw from that generative distribution. This computation is equivalent to calculating `sum((v - mean(v)).^2) / (length(v) - 1)`.

varm(*v*, *m*)
Compute the sample variance of a vector `v` with known mean `m`.

median(*v*)
Compute the median of a vector `v`.

hist(*v*[, *n*]) → e, counts
Compute the histogram of `v`, optionally using approximately `n` bins. The return values are a range `e`, which correspond to the edges of the bins, and `counts` containing the number of elements of `v` in each bin.

hist(*v*, *e*) → e, counts
Compute the histogram of `v` using a vector/range `e` as the edges for the bins. The result will be a vector of length `length(e) - 1`, with the `i''th element being ''sum(e[i] .< v .<= e[i+1])`.

histrange(*v*, *n*)
Compute *nice* bin ranges for the edges of a histogram of `v`, using approximately `n` bins. The resulting step sizes will be 1, 2 or 5 multiplied by a power of 10.

midpoints(*e*)
Compute the midpoints of the bins with edges `e`. The result is a vector/range of length `length(e) - 1`.

quantile(*v*, *p*)
Compute the quantiles of a vector `v` at a specified set of probability values `p`.

quantile(*v*)
Compute the quantiles of a vector `v` at the probability values `[.0, .2, .4, .6, .8, 1.0]`.

cov(*v1*[, *v2*])
Compute the Pearson covariance between two vectors `v1` and `v2`. If called with a single element `v`, then computes covariance of columns of `v`.

cor(*v1*[, *v2*])
Compute the Pearson correlation between two vectors `v1` and `v2`. If called with a single element `v`, then computes correlation of columns of `v`.

2.1.24 Signal Processing

FFT functions in Julia are largely implemented by calling functions from FFTW

fft(*A*[, *dims*])
Performs a multidimensional FFT of the array `A`. The optional `dims` argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. Most efficient if the size of `A` along the transformed dimensions is a product of small primes; see `nextprod()`. See also `plan_fft()` for even greater efficiency.

A one-dimensional FFT computes the one-dimensional discrete Fourier transform (DFT) as defined by $\mathrm{DFT}[k] = \sum_{n=1}^{\mathrm{length}(A)} \exp\left(-i\frac{2\pi(n-1)(k-1)}{\mathrm{length}(A)}\right) A[n]$. A multidimensional FFT simply performs this operation along each transformed dimension of `A`.

fft!(*A*[, *dims*])
Same as `fft()`, but operates in-place on `A`, which must be an array of complex floating-point numbers.

ifft(*A*[, *dims*])
Multidimensional inverse FFT.

A one-dimensional backward FFT computes $\mathrm{BDFT}[k] = \sum_{n=1}^{\mathrm{length}(A)} \exp\left(+i\frac{2\pi(n-1)(k-1)}{\mathrm{length}(A)}\right) A[n]$. A multidimensional backward FFT simply performs this operation along each transformed dimension of `A`. The inverse FFT computes the same thing divided by the product of the transformed dimensions.

ifft!(*A*[, *dims*])
Same as `ifft()`, but operates in-place on `A`.

bfft(*A*[, *dims*])
Similar to `ifft()`, but computes an unnormalized inverse (backward) transform, which must be divided by the product of the sizes of the transformed dimensions in order to obtain the inverse. (This is slightly more efficient than `ifft()` because it omits a scaling step, which in some applications can be combined with other computational steps elsewhere.)

bfft! (*A*[, *dims*])
Same as `bfft()`, but operates in-place on `A`.

plan_fft (*A*[, *dims*[, *flags*[, *timelimit*]]])
Pre-plan an optimized FFT along given dimensions (`dims`) of arrays matching the shape and type of `A`. (The first two arguments have the same meaning as for `fft()`.) Returns a function `plan(A)` that computes `fft(A, dims)` quickly.

The `flags` argument is a bitwise-or of FFTW planner flags, defaulting to `FFTW.ESTIMATE`. e.g. passing `FFTW.MEASURE` or `FFTW.PATIENT` will instead spend several seconds (or more) benchmarking different possible FFT algorithms and picking the fastest one; see the FFTW manual for more information on planner flags. The optional `timelimit` argument specifies a rough upper bound on the allowed planning time, in seconds. Passing `FFTW.MEASURE` or `FFTW.PATIENT` may cause the input array `A` to be overwritten with zeros during plan creation.

`plan_fft!()` is the same as `plan_fft()` but creates a plan that operates in-place on its argument (which must be an array of complex floating-point numbers). `plan_ifft()` and so on are similar but produce plans that perform the equivalent of the inverse transforms `ifft()` and so on.

plan_ifft (*A*[, *dims*[, *flags*[, *timelimit*]]])
Same as `plan_fft()`, but produces a plan that performs inverse transforms `ifft()`.

plan_bfft (*A*[, *dims*[, *flags*[, *timelimit*]]])
Same as `plan_fft()`, but produces a plan that performs an unnormalized backwards transform `bfft()`.

plan_fft! (*A*[, *dims*[, *flags*[, *timelimit*]]])
Same as `plan_fft()`, but operates in-place on `A`.

plan_ifft! (*A*[, *dims*[, *flags*[, *timelimit*]]])
Same as `plan_ifft()`, but operates in-place on `A`.

plan_bfft! (*A*[, *dims*[, *flags*[, *timelimit*]]])
Same as `plan_bfft()`, but operates in-place on `A`.

rfft (*A*[, *dims*])
Multidimensional FFT of a real array A, exploiting the fact that the transform has conjugate symmetry in order to save roughly half the computational time and storage costs compared with `fft()`. If `A` has size `(n_1, ..., n_d)`, the result has size `(floor(n_1/2)+1, ..., n_d)`.

The optional `dims` argument specifies an iterable subset of one or more dimensions of `A` to transform, similar to `fft()`. Instead of (roughly) halving the first dimension of `A` in the result, the `dims[1]` dimension is (roughly) halved in the same way.

irfft (*A*, *d*[, *dims*])
Inverse of `rfft()`: for a complex array `A`, gives the corresponding real array whose FFT yields `A` in the first half. As for `rfft()`, `dims` is an optional subset of dimensions to transform, defaulting to `1:ndims(A)`.

`d` is the length of the transformed real array along the `dims[1]` dimension, which must satisfy `d == floor(size(A,dims[1])/2)+1`. (This parameter cannot be inferred from `size(A)` due to the possibility of rounding by the `floor` function here.)

brfft (*A*, *d*[, *dims*])
Similar to `irfft()` but computes an unnormalized inverse transform (similar to `bfft()`), which must be divided by the product of the sizes of the transformed dimensions (of the real output array) in order to obtain the inverse transform.

plan_rfft (*A*[, *dims*[, *flags*[, *timelimit*]]])
Pre-plan an optimized real-input FFT, similar to `plan_fft()` except for `rfft()` instead of `fft()`. The first two arguments, and the size of the transformed result, are the same as for `rfft()`.

plan_irfft (*A*, *d*[, *dims*[, *flags*[, *timelimit*]]])
Pre-plan an optimized inverse real-input FFT, similar to `plan_rfft()` except for `irfft()` and `brfft()`, respectively. The first three arguments have the same meaning as for `irfft()`.

dct (*A*[, *dims*])
Performs a multidimensional type-II discrete cosine transform (DCT) of the array `A`, using the unitary normalization of the DCT. The optional `dims` argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. Most efficient if the size of `A` along the transformed dimensions is a product of small primes; see `nextprod()`. See also `plan_dct()` for even greater efficiency.

dct! (*A*[, *dims*])
Same as `dct!()`, except that it operates in-place on `A`, which must be an array of real or complex floating-point values.

idct (*A*[, *dims*])
Computes the multidimensional inverse discrete cosine transform (DCT) of the array `A` (technically, a type-III DCT with the unitary normalization). The optional `dims` argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. Most efficient if the size of `A` along the transformed dimensions is a product of small primes; see `nextprod()`. See also `plan_idct()` for even greater efficiency.

idct! (*A*[, *dims*])
Same as `idct!()`, but operates in-place on `A`.

plan_dct (*A*[, *dims*[, *flags*[, *timelimit*]]])
Pre-plan an optimized discrete cosine transform (DCT), similar to `plan_fft()` except producing a function that computes `dct()`. The first two arguments have the same meaning as for `dct()`.

plan_dct! (*A*[, *dims*[, *flags*[, *timelimit*]]])
Same as `plan_dct()`, but operates in-place on `A`.

plan_idct (*A*[, *dims*[, *flags*[, *timelimit*]]])
Pre-plan an optimized inverse discrete cosine transform (DCT), similar to `plan_fft()` except producing a function that computes `idct()`. The first two arguments have the same meaning as for `idct()`.

plan_idct! (*A*[, *dims*[, *flags*[, *timelimit*]]])
Same as `plan_idct()`, but operates in-place on `A`.

`FFTW.`**r2r** (*A*, *kind*[, *dims*])
Performs a multidimensional real-input/real-output (r2r) transform of type `kind` of the array `A`, as defined in the FFTW manual. `kind` specifies either a discrete cosine transform of various types (`FFTW.REDFT00`, `FFTW.REDFT01`, `FFTW.REDFT10`, or `FFTW.REDFT11`), a discrete sine transform of various types (`FFTW.RODFT00`, `FFTW.RODFT01`, `FFTW.RODFT10`, or `FFTW.RODFT11`), a real-input DFT with halfcomplex-format output (`FFTW.R2HC` and its inverse `FFTW.HC2R`), or a discrete Hartley transform (`FFTW.DHT`). The `kind` argument may be an array or tuple in order to specify different transform types along the different dimensions of `A`; `kind[end]` is used for any unspecified dimensions. See the FFTW manual for precise definitions of these transform types, at <*http://www.fftw.org/doc*>.

The optional `dims` argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. `kind[i]` is then the transform type for `dims[i]`, with `kind[end]` being used for `i > length(kind)`.

See also `FFTW.plan_r2r()` to pre-plan optimized r2r transforms.

`FFTW.`**r2r!** (*A*, *kind*[, *dims*])
`FFTW.r2r!()` is the same as `FFTW.r2r()`, but operates in-place on `A`, which must be an array of real or complex floating-point numbers.

`FFTW.`**plan_r2r** (*A*, *kind*[, *dims*[, *flags*[, *timelimit*]]])
Pre-plan an optimized r2r transform, similar to `plan_fft()` except that the transforms (and the first three

arguments) correspond to `FFTW.r2r()` and `FFTW.r2r!()`, respectively.

`FFTW.`**`plan_r2r!`**(*A*, *kind*[, *dims*[, *flags*[, *timelimit*]]])
Similar to `plan_fft()`, but corresponds to `FFTW.r2r!()`.

`fftshift`(*x*)
Swap the first and second halves of each dimension of `x`.

`fftshift`(*x*, *dim*)
Swap the first and second halves of the given dimension of array `x`.

`ifftshift`(*x*[, *dim*])
Undoes the effect of `fftshift`.

`filt`(*b*, *a*, *x*)
Apply filter described by vectors `a` and `b` to vector `x`.

`deconv`(*b*, *a*)
Construct vector `c` such that `b = conv(a,c) + r`. Equivalent to polynomial division.

`conv`(*u*, *v*)
Convolution of two vectors. Uses FFT algorithm.

`xcorr`(*u*, *v*)
Compute the cross-correlation of two vectors.

2.1.25 Parallel Computing

`addprocs_local`(*n*)
Add processes on the local machine. Can be used to take advantage of multiple cores.

`addprocs_ssh`(*{"host1", "host2", ...}*)
Add processes on remote machines via SSH. Requires julia to be installed in the same location on each node, or to be available via a shared file system.

`addprocs_sge`(*n*)
Add processes via the Sun/Oracle Grid Engine batch queue, using `qsub`.

`nprocs`()
Get the number of available processors.

`myid`()
Get the id of the current processor.

`pmap`(*f*, *c*)
Transform collection `c` by applying `f` to each element in parallel.

`remote_call`(*id*, *func*, *args...*)
Call a function asynchronously on the given arguments on the specified processor. Returns a `RemoteRef`.

`wait`(*RemoteRef*)
Wait for a value to become available for the specified remote reference.

`fetch`(*RemoteRef*)
Wait for and get the value of a remote reference.

`remote_call_wait`(*id*, *func*, *args...*)
Perform `wait(remote_call(...))` in one message.

`remote_call_fetch`(*id*, *func*, *args...*)
Perform `fetch(remote_call(...))` in one message.

put (*RemoteRef*, *value*)
Store a value to a remote reference. Implements "shared queue of length 1" semantics: if a value is already present, blocks until the value is removed with `take`.

take (*RemoteRef*)
Fetch the value of a remote reference, removing it so that the reference is empty again.

RemoteRef ()
Make an uninitialized remote reference on the local machine.

RemoteRef (*n*)
Make an uninitialized remote reference on processor `n`.

2.1.26 Distributed Arrays

DArray (*init*, *dims*[, *procs*, *dist*])
Construct a distributed array. `init` is a function accepting a tuple of index ranges. This function should return a chunk of the distributed array for the specified indexes. `dims` is the overall size of the distributed array. `procs` optionally specifies a vector of processor IDs to use. `dist` is an integer vector specifying how many chunks the distributed array should be divided into in each dimension.

dzeros (*dims*, ...)
Construct a distributed array of zeros. Trailing arguments are the same as those accepted by `darray`.

dones (*dims*, ...)
Construct a distributed array of ones. Trailing arguments are the same as those accepted by `darray`.

dfill (*x*, *dims*, ...)
Construct a distributed array filled with value `x`. Trailing arguments are the same as those accepted by `darray`.

drand (*dims*, ...)
Construct a distributed uniform random array. Trailing arguments are the same as those accepted by `darray`.

drandn (*dims*, ...)
Construct a distributed normal random array. Trailing arguments are the same as those accepted by `darray`.

distribute (*a*)
Convert a local array to distributed

localize (*d*)
Get the local piece of a distributed array

myindexes (*d*)
A tuple describing the indexes owned by the local processor

procs (*d*)
Get the vector of processors storing pieces of `d`

2.1.27 System

run (*command*)
Run a command object, constructed with backticks. Throws an error if anything goes wrong, including the process exiting with a non-zero status.

spawn (*command*)
Run a command object asynchronously, returning the resulting `Process` object.

success (*command*)
Run a command object, constructed with backticks, and tell whether it was successful (exited with a code of 0).

readsfrom(*command*)
Starts running a command asynchronously, and returns a tuple (stream,process). The first value is a stream reading from the process' standard output.

writesto(*command*)
Starts running a command asynchronously, and returns a tuple (stream,process). The first value is a stream writing to the process' standard input.

readandwrite(*command*)
Starts running a command asynchronously, and returns a tuple (stdout,stdin,process) of the output stream and input stream of the process, and the process object itself.

>()
Redirect standard output of a process.

Example: `run(`ls` > "out.log")`

<()
Redirect standard input of a process.

>>()
Redirect standard output of a process, appending to the destination file.

.>()
Redirect the standard error stream of a process.

gethostname() → String
Get the local machine's host name.

getipaddr() → String
Get the IP address of the local machine, as a string of the form "x.x.x.x".

pwd() → String
Get the current working directory.

cd(*dir::String*)
Set the current working directory. Returns the new current directory.

cd(*f*[, *"dir"*])
Temporarily changes the current working directory (HOME if not specified) and applies function f before returning.

mkdir(*path*[, *mode*])
Make a new directory with name `path` and permissions `mode`. `mode` defaults to 0o777, modified by the current file creation mask.

mkpath(*path*[, *mode*])
Create all directories in the given `path`, with permissions `mode`. `mode` defaults to 0o777, modified by the current file creation mask.

rmdir(*path*)
Remove the directory named `path`.

getpid() → Int32
Get julia's process ID.

time()
Get the system time in seconds since the epoch, with fairly high (typically, microsecond) resolution.

time_ns()
Get the time in nanoseconds. The time corresponding to 0 is undefined, and wraps every 5.8 years.

tic()
Set a timer to be read by the next call to `toc()` or `toq()`. The macro call `@time expr` can also be used to time evaluation.

toc()
Print and return the time elapsed since the last `tic()`.

toq()
Return, but do not print, the time elapsed since the last `tic()`.

EnvHash() → EnvHash
A singleton of this type provides a hash table interface to environment variables.

ENV
Reference to the singleton `EnvHash`, providing a dictionary interface to system environment variables.

2.1.28 C Interface

ccall (*(symbol*, *library) or fptr*, *RetType*, *(ArgType1*, *...)*, *ArgVar1*, *...*)
Call function in C-exported shared library, specified by (function name, library) tuple (String or :Symbol). Alternatively, ccall may be used to call a function pointer returned by dlsym, but note that this usage is generally discouraged to facilitate future static compilation.

cfunction (*fun::Function*, *RetType::Type*, *(ArgTypes...)*)
Generate C-callable function pointer from Julia function.

dlopen (*libfile::String*[, *flags::Integer*])
Load a shared library, returning an opaque handle.

The optional flags argument is a bitwise-or of zero or more of RTLD_LOCAL, RTLD_GLOBAL, RTLD_LAZY, RTLD_NOW, RTLD_NODELETE, RTLD_NOLOAD, RTLD_DEEPBIND, and RTLD_FIRST. These are converted to the corresponding flags of the POSIX (and/or GNU libc and/or MacOS) dlopen command, if possible, or are ignored if the specified functionality is not available on the current platform. The default is RTLD_LAZY|RTLD_DEEPBIND|RTLD_LOCAL. An important usage of these flags, on POSIX platforms, is to specify RTLD_LAZY|RTLD_DEEPBIND|RTLD_GLOBAL in order for the library's symbols to be available for usage in other shared libraries, in situations where there are dependencies between shared libraries.

dlsym (*handle*, *sym*)
Look up a symbol from a shared library handle, return callable function pointer on success.

dlsym_e (*handle*, *sym*)
Look up a symbol from a shared library handle, silently return NULL pointer on lookup failure.

dlclose (*handle*)
Close shared library referenced by handle.

c_free (*addr::Ptr*)
Call free() from C standard library.

unsafe_ref (*p::Ptr{T}*, *i::Integer*)
Dereference the pointer `p[i]` or `*p`, returning a copy of type T.

unsafe_assign (*p::Ptr{T}*, *x*, *i::Integer*)
Assign to the pointer `p[i] = x` or `*p = x`, making a copy of object x into the memory at p.

pointer (*a*[, *index*])
Get the native address of an array element. Be careful to ensure that a julia reference to `a` exists as long as this pointer will be used.

pointer (*type*, *int*)
Convert an integer to a pointer of the specified element type.

pointer_to_array(*p*, *dims*[, *own*])
Wrap a native pointer as a Julia Array object. The pointer element type determines the array element type. `own` optionally specifies whether Julia should take ownership of the memory, calling `free` on the pointer when the array is no longer referenced.

2.1.29 Errors

error(*message::String*)
Raise an error with the given message

throw(*e*)
Throw an object as an exception

errno()
Get the value of the C library's `errno`

strerror(*n*)
Convert a system call error code to a descriptive string

assert(*cond*)
Raise an error if `cond` is false. Also available as the macro `@assert expr`.

2.1.30 Tasks

Task(*func*)
Create a `Task` (i.e. thread, or coroutine) to execute the given function. The task exits when this function returns.

yieldto(*task*, *args...*)
Switch to the given task. The first time a task is switched to, the task's function is called with `args`. On subsequent switches, `args` are returned from the task's last call to `yieldto`.

current_task()
Get the currently running Task.

istaskdone(*task*)
Tell whether a task has exited.

consume(*task*)
Receive the next value passed to `produce` by the specified task.

produce(*value*)
Send the given value to the last `consume` call, switching to the consumer task.

make_scheduled(*task*)
Register a task with the main event loop, so it will automatically run when possible.

yield()
For scheduled tasks, switch back to the scheduler to allow another scheduled task to run.

tls(*symbol*)
Look up the value of a symbol in the current task's task-local storage.

tls(*symbol*, *value*)
Assign a value to a symbol in the current task's task-local storage.

2.1.31 Sparse Matrices

Sparse matrices support much of the same set of operations as dense matrices. The following functions are specific to sparse matrices.

sparse(*I*, *J*, *V*[, *m*, *n*, *combine*])
Create a sparse matrix `S` of dimensions `m x n` such that `S[I[k], J[k]] = V[k]`. The `combine` function is used to combine duplicates. If `m` and `n` are not specified, they are set to `max(I)` and `max(J)` respectively. If the `combine` function is not supplied, duplicates are added by default.

sparsevec(*I*, *V*[, *m*, *combine*])
Create a sparse matrix `S` of size `m x 1` such that `S[I[k]] = V[k]`. Duplicates are combined using the `combine` function, which defaults to + if it is not provided. In julia, sparse vectors are really just sparse matrices with one column. Given Julia's Compressed Sparse Columns (CSC) storage format, a sparse column matrix with one column is sparse, whereas a sparse row matrix with one row ends up being dense.

sparsevec(*D::Dict*[, *m*])
Create a sparse matrix of size `m x 1` where the row values are keys from the dictionary, and the nonzero values are the values from the dictionary.

issparse(*S*)
Returns `true` if `S` is sparse, and `false` otherwise.

sparse(*A*)
Convert a dense matrix `A` into a sparse matrix.

sparsevec(*A*)
Convert a dense vector `A` into a sparse matrix of size `m x 1`. In julia, sparse vectors are really just sparse matrices with one column.

dense(*S*)
Convert a sparse matrix `S` into a dense matrix.

full(*S*)
Convert a sparse matrix `S` into a dense matrix.

spzeros(*m*, *n*)
Create an empty sparse matrix of size `m x n`.

speye(*type*, *m*[, *n*])
Create a sparse identity matrix of specified type of size `m x m`. In case `n` is supplied, create a sparse identity matrix of size `m x n`.

spones(*S*)
Create a sparse matrix with the same structure as that of `S`, but with every nonzero element having the value `1.0`.

sprand(*m*, *n*, *density*[, *rng*])
Create a random sparse matrix with the specified density. Nonzeros are sampled from the distribution specified by `rng`. The uniform distribution is used in case `rng` is not specified.

sprandn(*m*, *n*, *density*)
Create a random sparse matrix of specified density with nonzeros sampled from the normal distribution.

sprandbool(*m*, *n*, *density*)
Create a random sparse boolean matrix with the specified density.

2.1.32 Linear Algebra

Linear algebra functions in Julia are largely implemented by calling functions from LAPACK. Sparse factorizations call functions from SuiteSparse.

***** (*A*, *B*)
Matrix multiplication

**** (*A*, *B*)
Matrix division using a polyalgorithm. For input matrices `A` and `B`, the result `X` is such that `A*X == B` when `A` is square. The solver that is used depends upon the structure of `A`. A direct solver is used for upper- or lower triangular `A`. For Hermitian `A` (equivalent to symmetric `A` for non-complex `A`) the BunchKaufman factorization is used. Otherwise an LU factorization is used. For rectangular `A` the result is the minimum-norm least squares solution computed by reducing `A` to bidiagonal form and solving the bidiagonal least squares problem. For sparse, square `A` the LU factorization (from UMFPACK) is used.

dot (*x*, *y*)
Compute the dot product

cross (*x*, *y*)
Compute the cross product of two 3-vectors

norm (*a*)
Compute the norm of a `Vector` or a `Matrix`

lu (*A*) → L, U, P
Compute the LU factorization of `A`, such that `P*A = L*U`.

lufact (*A*) → LU
Compute the LU factorization of `A`, returning an `LU` object for dense `A` or an `UmfpackLU` object for sparse `A`. The individual components of the factorization `F` can be accesed by indexing: `F[:L]`, `F[:U]`, and `F[:P]` (permutation matrix) or `F[:p]` (permutation vector). An `UmfpackLU` object has additional components `F[:q]` (the left permutation vector) and `Rs` the vector of scaling factors. The following functions are available for both `LU` and `UmfpackLU` objects: `size`, `\` and `det`. For `LU` there is also an `inv` method. The sparse LU factorization is such that `L*U` is equal to‘‘diagmm(Rs,A)[p,q]‘‘.

lufact! (*A*) → LU
`lufact!` is the same as `lufact` but saves space by overwriting the input A, instead of creating a copy. For sparse `A` the `nzval` field is not overwritten but the index fields, `colptr` and `rowval` are decremented in place, converting from 1-based indices to 0-based indices.

chol (*A*[, *LU*]) → F
Compute Cholesky factorization of a symmetric positive-definite matrix `A` and return the matrix `F`. If `LU` is `L` (Lower), `A = L*L'`. If `LU` is `U` (Upper), `A = R'*R`.

cholfact (*A*[, *LU*]) → Cholesky
Compute the Cholesky factorization of a dense symmetric positive-definite matrix `A` and return a `Cholesky` object. `LU` may be ‘L’ for using the lower part or ‘U’ for the upper part. The default is to use ‘U’. The triangular matrix can be obtained from the factorization `F` with: `F[:L]` and `F[:U]`. The following functions are available for `Cholesky` objects: `size`, `\`, `inv`, `det`. A `LAPACK.PosDefException` error is thrown in case the matrix is not positive definite.

cholfact (*A*[, *ll*]) → CholmodFactor
Compute the sparse Cholesky factorization of a sparse matrix `A`. If `A` is Hermitian its Cholesky factor is determined. If `A` is not Hermitian the Cholesky factor of `A*A'` is determined. A fill-reducing permutation is used. Methods for `size`, `solve`, `\`, `findn_nzs`, `diag`, `det` and `logdet`. One of the solve methods includes an integer argument that can be used to solve systems involving parts of the factorization only. The optional boolean argument, `ll` determines whether the factorization returned is of the `A[p,p] = L*L'` form, where

`L` is lower triangular or `A[p,p] = diagmm(L,D)*L'` form where `L` is unit lower triangular and `D` is a non-negative vector. The default is LDL.

`cholfact!` (*A*[, *LU*]) → Cholesky
`cholfact!` is the same as `cholfact` but saves space by overwriting the input A, instead of creating a copy.

`cholpfact` (*A*[, *LU*]) → CholeskyPivoted
Compute the pivoted Cholesky factorization of a symmetric positive semi-definite matrix `A` and return a `CholeskyPivoted` object. `LU` may be 'L' for using the lower part or 'U' for the upper part. The default is to use 'U'. The triangular factors containted in the factorization `F` can be obtained with `F[:L]` and `F[:U]`, whereas the permutation can be obtained with `F[:P]` or `F[:p]`. The following functions are available for `CholeskyPivoted` objects: `size`, `\`, `inv`, `det`. A `LAPACK.RankDeficientException` error is thrown in case the matrix is rank deficient.

`cholpfact!` (*A*[, *LU*]) → CholeskyPivoted
`cholpfact!` is the same as `cholpfact` but saves space by overwriting the input A, instead of creating a copy.

`qr` (*A*[, *thin*]) → Q, R
Compute the QR factorization of `A` such that `A = Q*R`. Also see `qrfact`. The default is to compute a thin factorization.

`qrfact` (*A*)
Compute the QR factorization of `A` and return a `QR` object. The coomponents of the factorization `F` can be accessed as follows: the orthogonal matrix `Q` can be extracted with `F[:Q]` and the triangular matrix `R` with `F[:R]`. The following functions are available for `QR` objects: `size`, `\`. When `Q` is extracted, the resulting type is the `QRPackedQ` object, and has the `*` operator overloaded to support efficient multiplication by `Q` and `Q'`.

`qrfact!` (*A*)
`qrfact!` is the same as `qrfact` but saves space by overwriting the input A, instead of creating a copy.

`qrp` (*A*[, *thin*]) → Q, R, P
Compute the QR factorization of `A` with pivoting, such that `A*P = Q*R`, Also see `qrpfact`. The default is to compute a thin factorization.

`qrpfact` (*A*) → QRPivoted
Compute the QR factorization of `A` with pivoting and return a `QRPivoted` object. The components of the factorization `F` can be accessed as follows: the orthogonal matrix `Q` can be extracted with `F[:Q]`, the triangular matrix `R` with `F[:R]`, and the permutation with `F[:P]` or `F[:p]`. The following functions are available for `QRPivoted` objects: `size`, `\`. When `Q` is extracted, the resulting type is the `QRPivotedQ` object, and has the `*` operator overloaded to support efficient multiplication by `Q` and `Q'`. A `QRPivotedQ` matrix can be converted into a regular matrix with `full`.

`qrpfact!` (*A*) → QRPivoted
`qrpfact!` is the same as `qrpfact` but saves space by overwriting the input A, instead of creating a copy.

`sqrtm` (*A*)
Compute the matrix square root of `A`. If `B = sqrtm(A)`, then `B*B == A` within roundoff error.

`eig` (*A*) → D, V
Compute eigenvalues and eigenvectors of A

`eigvals` (*A*)
Returns the eigenvalues of `A`.

`eigmax` (*A*)
Returns the largest eigenvalue of `A`.

`eigmin` (*A*)
Returns the smallest eigenvalue of `A`.

eigvecs (*A*[, *eigvals*])
Returns the eigenvectors of A.

For SymTridiagonal matrices, if the optional vector of eigenvalues eigvals is specified, returns the specific corresponding eigenvectors.

eigfact (*A*)
Compute the eigenvalue decomposition of A and return an Eigen object. If F is the factorization object, the eigenvalues can be accessed with F[:values] and the eigenvectors with F[:vectors]. The following functions are available for Eigen objects: inv, det.

eigfact! (*A*)
eigfact! is the same as eigfact but saves space by overwriting the input A, instead of creating a copy.

hessfact (*A*)
Compute the Hessenberg decomposition of A and return a Hessenberg object. If F is the factorization object, the unitary matrix can be accessed with F[:Q] and the Hessenberg matrix with F[:H]. When Q is extracted, the resulting type is the HessenbergQ object, and may be converted to a regular matrix with full.

hessfact! (*A*)
hessfact! is the same as hessfact but saves space by overwriting the input A, instead of creating a copy.

schurfact (*A*) → Schur
Computes the Schur factorization of the matrix A. The (quasi) triangular Schur factor can be obtained from the Schur object F with either F[:Schur] or F[:T] and the unitary/orthogonal Schur vectors can be obtained with F[:vectors] or F[:Z] such that A=F[:vectors]*F[:Schur]*F[:vectors]'. The eigenvalues of A can be obtained with F[:values].

schur (*A*) → Schur[:T], Schur[:Z], Schur[:values]
See schurfact

schurfact (*A*, *B*) → GeneralizedSchur
Computes the Generalized Schur (or QZ) factorization of the matrices A and B. The (quasi) triangular Schur factors can be obtained from the Schur object F with F[:S] and F[:T], the left unitary/orthogonal Schur vectors can be obtained with F[:left] or F[:Q] and the right unitary/orthogonal Schur vectors can be obtained with F[:right] or F[:Z] such that A=F[:left]*F[:S]*F[:right]' and B=F[:left]*F[:T]*F[:right]'. The generalized eigenvalues of A and B can be obtained with F[:alpha]./F[:beta].

schur (*A*, *B*) → GeneralizedSchur[:S], GeneralizedSchur[:T], GeneralizedSchur[:Q], GeneralizedSchur[:Z]
See schurfact

svdfact (*A*[, *thin*]) → SVD
Compute the Singular Value Decomposition (SVD) of A and return an SVD object. U, S, V and Vt can be obtained from the factorization F with F[:U], F[:S], F[:V] and F[:Vt], such that A = U*diagm(S)*Vt. If thin is true, an economy mode decomposition is returned. The algorithm produces Vt and hence Vt is more efficient to extract than V. The default is to produce a thin decomposition.

svdfact! (*A*[, *thin*]) → SVD
svdfact! is the same as svdfact but saves space by overwriting the input A, instead of creating a copy. If thin is true, an economy mode decomposition is returned. The default is to produce a thin decomposition.

svd (*A*[, *thin*]) → U, S, V
Compute the SVD of A, returning U, vector S, and V such that A == U*diagm(S)*V'. If thin is true, an economy mode decomposition is returned.

svdvals (*A*)
Returns the singular values of A.

svdvals! (*A*)
Returns the singular values of A, while saving space by overwriting the input.

svdfact(*A*, *B*) → GeneralizedSVD
Compute the generalized SVD of `A` and `B`, returning a `GeneralizedSVD` Factorization object, such that `A = U*D1*R0*Q'` and `B = V*D2*R0*Q'`.

svd(*A*, *B*) → U, V, Q, D1, D2, R0
Compute the generalized SVD of `A` and `B`, returning `U`, `V`, `Q`, `D1`, `D2`, and `R0` such that `A = U*D1*R0*Q'` and `B = V*D2*R0*Q'`.

svdvals(*A*, *B*)
Return only the singular values from the generalized singular value decomposition of `A` and `B`.

triu(*M*)
Upper triangle of a matrix

tril(*M*)
Lower triangle of a matrix

diag(*M*[, *k*])
The `k`-th diagonal of a matrix, as a vector

diagm(*v*[, *k*])
Construct a diagonal matrix and place `v` on the `k`-th diagonal

diagmm(*matrix*, *vector*)
Multiply matrices, interpreting the vector argument as a diagonal matrix. The arguments may occur in the other order to multiply with the diagonal matrix on the left.

Tridiagonal(*dl*, *d*, *du*)
Construct a tridiagonal matrix from the lower diagonal, diagonal, and upper diagonal

Bidiagonal(*dv*, *ev*, *isupper*)
Constructs an upper (isupper=true) or lower (isupper=false) bidiagonal matrix using the given diagonal (dv) and off-diagonal (ev) vectors

Woodbury(*A*, *U*, *C*, *V*)
Construct a matrix in a form suitable for applying the Woodbury matrix identity

rank(*M*)
Compute the rank of a matrix

norm(*A*[, *p*])
Compute the `p`-norm of a vector or a matrix. `p` is `2` by default, if not provided. If `A` is a vector, `norm(A, p)` computes the `p`-norm. `norm(A, Inf)` returns the largest value in `abs(A)`, whereas `norm(A, -Inf)` returns the smallest. If `A` is a matrix, valid values for `p` are `1`, `2`, or `Inf`. In order to compute the Frobenius norm, use `normfro`.

normfro(*A*)
Compute the Frobenius norm of a matrix `A`.

cond(*M*[, *p*])
Matrix condition number, computed using the p-norm. `p` is 2 by default, if not provided. Valid values for `p` are `1`, `2`, or `Inf`.

trace(*M*)
Matrix trace

det(*M*)
Matrix determinant

inv(*M*)
Matrix inverse

pinv(*M*)
Moore-Penrose inverse

null(*M*)
Basis for null space of M.

repmat(*A*, *n*, *m*)
Construct a matrix by repeating the given matrix `n` times in dimension 1 and `m` times in dimension 2.

kron(*A*, *B*)
Kronecker tensor product of two vectors or two matrices.

linreg(*x*, *y*)
Determine parameters `[a, b]` that minimize the squared error between `y` and `a+b*x`.

linreg(*x*, *y*, *w*)
Weighted least-squares linear regression.

expm(*A*)
Matrix exponential.

issym(*A*)
Test whether a matrix is symmetric.

isposdef(*A*)
Test whether a matrix is positive-definite.

istril(*A*)
Test whether a matrix is lower-triangular.

istriu(*A*)
Test whether a matrix is upper-triangular.

ishermitian(*A*)
Test whether a matrix is hermitian.

transpose(*A*)
The transpose operator (.').

ctranspose(*A*)
The conjugate transpose operator (').

2.1.33 BLAS Functions

This module provides wrappers for some of the BLAS functions for linear algebra. Those BLAS functions that overwrite one of the input arrays have names ending in `'!'`.

Usually a function has 4 methods defined, one each for `Float64`, `Float32`, `Complex128` and `Complex64` arrays.

copy!(*n*, *X*, *incx*, *Y*, *incy*)
Copy `n` elements of array `X` with stride `incx` to array `Y` with stride `incy`. Returns `Y`.

dot(*n*, *X*, *incx*, *Y*, *incy*)
Dot product of two vectors consisting of `n` elements of array `X` with stride `incx` and `n` elements of array `Y` with stride `incy`. There are no `dot` methods for `Complex` arrays.

nrm2(*n*, *X*, *incx*)
2-norm of a vector consisting of `n` elements of array `X` with stride `incx`.

axpy!(*n*, *a*, *X*, *incx*, *Y*, *incy*)
Overwrite `Y` with `a*X + Y`. Returns `Y`.

syrk! (*uplo*, *trans*, *alpha*, *A*, *beta*, *C*)
Rank-k update of the symmetric matrix `C` as `alpha*A*A.' + beta*C` or `alpha*A.'*A + beta*C` according to whether `trans` is 'N' or 'T'. When `uplo` is 'U' the upper triangle of `C` is updated ('L' for lower triangle). Returns `C`.

syrk (*uplo*, *trans*, *alpha*, *A*)
Returns either the upper triangle or the lower triangle, according to `uplo` ('U' or 'L'), of `alpha*A*A.'` or `alpha*A.'*A`, according to `trans` ('N' or 'T').

herk! (*uplo*, *trans*, *alpha*, *A*, *beta*, *C*)
Methods for complex arrays only. Rank-k update of the Hermitian matrix `C` as `alpha*A*A' + beta*C` or `alpha*A'*A + beta*C` according to whether `trans` is 'N' or 'T'. When `uplo` is 'U' the upper triangle of `C` is updated ('L' for lower triangle). Returns `C`.

herk (*uplo*, *trans*, *alpha*, *A*)
Methods for complex arrays only. Returns either the upper triangle or the lower triangle, according to `uplo` ('U' or 'L'), of `alpha*A*A'` or `alpha*A'*A`, according to `trans` ('N' or 'T').

gbmv! (*trans*, *m*, *kl*, *ku*, *alpha*, *A*, *x*, *beta*, *y*)
Update vector `y` as `alpha*A*x + beta*y` or `alpha*A'*x + beta*y` according to `trans` ('N' or 'T'). The matrix `A` is a general band matrix of dimension `m` by `size(A,2)` with `kl` sub-diagonals and `ku` super-diagonals. Returns the updated `y`.

gbmv (*trans*, *m*, *kl*, *ku*, *alpha*, *A*, *x*, *beta*, *y*)
Returns `alpha*A*x` or `alpha*A'*x` according to `trans` ('N' or 'T'). The matrix `A` is a general band matrix of dimension `m` by `size(A,2)` with `kl` sub-diagonals and `ku` super-diagonals.

sbmv! (*uplo*, *k*, *alpha*, *A*, *x*, *beta*, *y*)
Update vector `y` as `alpha*A*x + beta*y` where `A` is a a symmetric band matrix of order `size(A,2)` with `k` super-diagonals stored in the argument `A`. The storage layout for `A` is described the reference BLAS module, level-2 BLAS at <*http://www.netlib.org/lapack/explore-html/*>.

Returns the updated `y`.

sbmv (*uplo*, *k*, *alpha*, *A*, *x*)
Returns `alpha*A*x` where `A` is a symmetric band matrix of order `size(A,2)` with `k` super-diagonals stored in the argument `A`.

gemm! (*tA*, *tB*, *alpha*, *A*, *B*, *beta*, *C*)
Update `C` as `alpha*A*B + beta*C` or the other three variants according to `tA` (transpose `A`) and `tB`. Returns the updated `C`.

gemm (*tA*, *tB*, *alpha*, *A*, *B*)
Returns `alpha*A*B` or the other three variants according to `tA` (transpose `A`) and `tB`.

2.1.34 Constants

OS_NAME
A symbol representing the name of the operating system. Possible values are `:Linux`, `:Darwin` (OS X), or `:Windows`.

ARGS
An array of the command line arguments passed to Julia, as strings.

C_NULL
The C null pointer constant, sometimes used when calling external code.

CPU_CORES
The number of CPU cores in the system.

WORD_SIZE
Standard word size on the current machine, in bits.

VERSION
An object describing which version of Julia is in use.

LOAD_PATH
An array of paths (as strings) where the `require` function looks for code.

2.1.35 Filesystem

isblockdev(*path*) → Bool
Returns `true` if `path` is a block device, `false` otherwise.

ischardev(*path*) → Bool
Returns `true` if `path` is a character device, `false` otherwise.

isdir(*path*) → Bool
Returns `true` if `path` is a directory, `false` otherwise.

isexecutable(*path*) → Bool
Returns `true` if the current user has permission to execute `path`, `false` otherwise.

isfifo(*path*) → Bool
Returns `true` if `path` is a FIFO, `false` otherwise.

isfile(*path*) → Bool
Returns `true` if `path` is a regular file, `false` otherwise.

islink(*path*) → Bool
Returns `true` if `path` is a symbolic link, `false` otherwise.

ispath(*path*) → Bool
Returns `true` if `path` is a valid filesystem path, `false` otherwise.

isreadable(*path*) → Bool
Returns `true` if the current user has permission to read `path`, `false` otherwise.

issetgid(*path*) → Bool
Returns `true` if `path` has the setgid flag set, `false` otherwise.

issetuid(*path*) → Bool
Returns `true` if `path` has the setuid flag set, `false` otherwise.

issocket(*path*) → Bool
Returns `true` if `path` is a socket, `false` otherwise.

issticky(*path*) → Bool
Returns `true` if `path` has the sticky bit set, `false` otherwise.

iswriteable(*path*) → Bool
Returns `true` if the current user has permission to write to `path`, `false` otherwise.

dirname(*path::String*) → String
Get the directory part of a path.

basename(*path::String*) → String
Get the file name part of a path.

isabspath(*path::String*) → Bool
Determines whether a path is absolute (begins at the root directory).

joinpath(*parts...*) → String
Join path components into a full path. If some argument is an absolute path, then prior components are dropped.

abspath(*path::String*) → String
Convert a path to an absolute path by adding the current directory if necessary.

tempname()
Generate a unique temporary filename.

tempdir()
Obtain the path of a temporary directory.

mktemp()
Returns `(path, io)`, where `path` is the path of a new temporary file and `io` is an open file object for this path.

mktempdir()
Create a temporary directory and return its path.

2.1.36 Punctuation

punctuation

symbol	meaning
`@m`	invoke macro m; followed by space-separated expressions
`!`	prefix "not" operator
`!`	at the end of a function name, indicates that a function modifies its argument(s)
`#`	begin single line comment
`$`	xor operator, string and expression interpolation
`%`	remainder operator
`^`	exponent operator
`&`	bitwise and
`*`	multiply, or matrix multiply
`()`	the empty tuple
`~`	bitwise not operator
`\`	backslash operator
`a[]`	array indexing
`[,]`	vertical concatenation
`[;]`	also vertical concatenation
`[ ]`	with space-separated expressions, horizontal concatenation
`T{ }`	parametric type instantiation
`{ }`	construct a cell array
`;`	statement separator
`,`	separate function arguments or tuple components
`?`	3-argument conditional operator
`""`	delimit string literals
`''`	delimit character literals
`` `` ```	delimit external process (command) specifications
`...`	splice arguments into a function call, or declare a varargs function
`.`	access named fields in objects or names inside modules, also prefixes elementwise operators
`a:b`	range
`a:s:b`	range
`:`	index an entire dimension
`::`	type annotation

Continuação na próxima página

Tabela 2.1 – continuação da página anterior

symbol	meaning
`:( )`	quoted expression

2.2 Built-in Modules

2.2.1 `Base.Sort` — Routines related to sorting

The *Sort* module contains algorithms and other functions related to sorting. Default sort functions and standard versions of the various sort algorithm are available by default. Specific sort algorithms can be used by importing *Sort* or using the fully qualified algorithm name, e.g.,:

```
# Julia code
sort(v, Sort.TimSort)
```

will sort `v` using `TimSort`.

Overview

Many users will simply want to use the default sort algorithms, which allow sorting in ascending or descending order,:

```
# Julia code
julia> sort([2,3,1]) == [1,2,3]
true

julia> sort([2,3,1], Sort.Reverse) == [3,2,1]
true
```

return a permutation,:

```
julia> v = [20,30,10]
3-element Int64 Array:
 20
 30
 10

julia> p = sortperm(v)
[3, 1, 2]

julia> v[p]
3-element Int64 Array:
 10
 20
 30
```

and use a custom extractor function to order inputs:

```
julia> canonicalize(s) = filter(c -> ('A'<=c<='Z' || 'a'<=c<='z'), s) | uppercase

julia> sortby(["New York", "New Jersey", "Nevada", "Nebraska", "Newark"], canonicalize)
5-element ASCIIString Array:
 "Nebraska"
 "Nevada"
 "Newark"
 "New Jersey"
 "New York"
```

Note that none of the variants above modify the original arrays. To sort in-place (which is often more efficient), `sort()` and `sortby()` have mutating versions which end with an exclamation point (`sort!()` and `sortby!()`).

These sort functions use reasonable default algorithms, but if you want more control or want to see if a different sort algorithm will work better on your data, read on...

Sort Algorithms

There are currently four main sorting algorithms available in Julia:

```
InsertionSort
QuickSort
MergeSort
TimSort
```

Insertion sort is an O(n^2) stable sorting algorithm. It is efficient for very small `n`, and is used internally by `QuickSort` and `TimSort`.

Quicksort is an O(n log n) sorting algorithm. For efficiency, it is not stable. It is among the fastest sorting algorithms.

Mergesort is an O(n log n) stable sorting algorithm.

Timsort is an O(n log n) stable adaptive sorting algorithm. It takes advantage of sorted runs which exist in many real world datasets.

The sort functions select a reasonable default algorithm, depending on the type of the target array. To force a specific algorithm to be used, append `Sort.<algorithm>` to the argument list (e.g., use `sort!(v, Sort.TimSort)` to force the use of the Timsort algorithm).

Functions

Sort Functions

`sort`(*v*[, *alg*[, *ord*]])
: Sort a vector in ascending order. Specify `alg` to choose a particular sorting algorithm (`Sort.InsertionSort`, `Sort.QuickSort`, `Sort.MergeSort`, or `Sort.TimSort`), and `ord` to sort with a custom ordering (e.g., Sort.Reverse or a comparison function).

`sort!`(...)
: In-place sort.

`sortby`(*v*, *by*[, *alg*])
: Sort a vector according to `by(v)`. Specify `alg` to choose a particular sorting algorithm (`Sort.InsertionSort`, `Sort.QuickSort`, `Sort.MergeSort`, or `Sort.TimSort`).

`sortby!`(...)
: In-place `sortby`.

`sortperm`(*v*[, *alg*[, *ord*]])
: Return a permutation vector, which when applied to the input vector `v` will sort it. Specify `alg` to choose a particular sorting algorithm (`Sort.InsertionSort`, `Sort.QuickSort`, `Sort.MergeSort`, or `Sort.TimSort`), and `ord` to sort with a custom ordering (e.g., Sort.Reverse or a comparison function).

Sorting-related Functions

`issorted`(*v*[, *ord*])
: Test whether a vector is in ascending sorted order. If specified, `ord` gives the ordering to test.

searchsorted(*a*, *x*[, *ord*])
Returns the index of the first value of `a` equal to or succeeding `x`, according to ordering `ord` (default: `Sort.Forward`).

Alias for `searchsortedfirst()`

searchsortedfirst(*a*, *x*[, *ord*])
Returns the index of the first value of `a` equal to or succeeding `x`, according to ordering `ord` (default: `Sort.Forward`).

searchsortedlast(*a*, *x*[, *ord*])
Returns the index of the last value of `a` preceding or equal to `x`, according to ordering `ord` (default: `Sort.Forward`).

select(*v*, *k*[, *ord*])
Find the element in position `k` in the sorted vector `v` without sorting, according to ordering `ord` (default: `Sort.Forward`).

select!(*v*, *k*[, *ord*])
Version of `select` which permutes the input vector in place.

Pacotes Disponíveis (em inglês)

3.1 ArgParse

Current Version: `0.2.0`

Package for parsing command-line arguments to Julia programs.

Maintainer: Carlo Baldassi

Dependencies:

```
Options         Any Version
TextWrap        Any Version
julia           [v"0.2.0-"]
```

Contributors:

3.2 Benchmark

Current Version: `0.0.0`

A package for benchmarking code and packages

Maintainer: John Myles White

Dependencies:

```
DataFrames      Any Version
```

Contributors:

3.3 BinDeps

Current Version: `0.0.0`

Tool for building binary dependencies for Julia modules

Maintainer: Keno Fischer

Dependencies:

None

Contributors:

3.4 BioSeq

Current Version: `0.0.0`

Julia's package for working on Bioinformatics with DNA, RNA and Protein Sequences

Maintainer: Diego Javier Zea

Dependencies:

None

Contributors:

3.5 BloomFilters

Current Version: `0.0.0`

Bloom filters in Julia

Maintainer: John Myles White

Dependencies:

None

Contributors:

3.6 Cairo

Current Version: `0.0.0`

Bindings to the Cairo graphics library.

Maintainer: The Julia Language

Dependencies:

```
BinDeps          Any Version
Color            Any Version
```

Contributors:

3.7 Calculus

Current Version: `0.0.0`

Calculus functions in Julia

Maintainer: John Myles White

Dependencies:

None

Contributors:

3.8 Calendar

Current Version: `0.0.0`

Calendar time package for Julia

Maintainer: Mike Nolta

Dependencies:

```
ICU              Any Version
```

Contributors:

3.9 Catalan

Current Version: `0.0.0`

Catalan: a combinatorics library for Julia

Maintainer: Alessandro Andrioni

Dependencies:

```
Polynomial      Any Version
```

Contributors:

3.10 Clang

Current Version: `0.0.0`

Julia access to the libclang interface of the LLVM Clang compiler.

Maintainer: Isaiah

Dependencies:

```
BinDeps         Any Version
julia           [v"0.2.0-"]
```

Contributors:

3.11 Clp

Current Version: `0.0.0`

Interface to the Coin-OR Linear Programming solver (CLP)

Maintainer: Miles Lubin

Dependencies:

```
BinDeps         Any Version
julia           [v"0.1.0-"]
```

Contributors:

3.12 Clustering

Current Version: `0.0.0`

Basic functions for clustering data: k-means, dp-means, etc.

Maintainer: John Myles White

Dependencies:

```
Devectorize     Any Version
Distance        Any Version
MLBase          Any Version
Options         Any Version
```

Contributors:

3.13 Codecs

Current Version: `0.0.0`

Common data encoding algorithms

Maintainer: Daniel Jones

Dependencies:

```
Iterators       Any Version
```

Contributors:

3.14 CoinMP

Current Version: `0.0.0`

Interface to the Coin-OR CBC solver for mixed-integer programming

Maintainer: Miles Lubin

Dependencies:

```
BinDeps         Any Version
```

Contributors:

3.15 Color

Current Version: `0.2.0`

Basic color manipulation utilities.

Maintainer: The Julia Language

Dependencies:

```
julia           [v"0.2.0-"]
```

Contributors:

3.16 Compose

Current Version: `0.0.0`

Declarative vector graphics

Maintainer: Daniel Jones

Dependencies:

```
Cairo           Any Version
Mustache        Any Version
```

Contributors:

3.17 ContinuedFractions

Current Version: `0.0.0`

Types and functions for working with continued fractions in Julia

Maintainer: John Myles White

Dependencies:

None

Contributors:

3.18 Cpp

Current Version: `0.0.0`

Utilities for calling C++ from Julia

Maintainer: Tim Holy

Dependencies:

None

Contributors:

3.19 Cubature

Current Version: `0.0.0`

One- and multi-dimensional adaptive integration routines for the Julia language

Maintainer: Steven G. Johnson

Dependencies:

`BinDeps` **Any** `Version`

Contributors:

3.20 Curl

Current Version: `0.0.0`

a Julia HTTP curl library

Maintainer: Forio Online Simulations

Dependencies:

None

Contributors:

3.21 DICOM

Current Version: `0.0.0`

DICOM for Julia

Maintainer: Isaiah

Dependencies:

None

Contributors:

3.22 DataFrames

Current Version: `0.2.0`

library for working with tabular data in Julia

Maintainer: Harlan Harris

Dependencies:

```
GZip            Any Version
Options         Any Version
Stats           Any Version
julia           [v"0.2.0-"]
```

Contributors:

3.23 DataStructures

Current Version: `0.0.0`

Julia implementation of Data structures

Maintainer: Dahua Lin

Dependencies:

None

Contributors:

3.24 Debug

Current Version: `0.0.0`

Prototype interactive debugger for Julia

Maintainer: toivoh

Dependencies:

None

Contributors:

3.25 DecisionTree

Current Version: `0.0.0`

Decision Tree Classifier in Julia

Maintainer: Ben Sadeghi

Dependencies:

```
julia           [v"0.1.0-"]
```

Contributors:

3.26 Devectorize

Current Version: `0.2.0`

A Julia framework for delayed expression evaluation

Maintainer: Dahua Lin

Dependencies:

```
julia           [v"0.2.0-"]
```

Contributors:

3.27 DictViews

Current Version: `0.0.0`

KeysView and ValuesView types for dynamic low-overhead views into the entries of dictionaries

Maintainer: David de Laat

Dependencies:

```
None
```

Contributors:

3.28 DimensionalityReduction

Current Version: `0.0.0`

Methods for dimensionality reduction: PCA, ICA, NMF

Maintainer: John Myles White

Dependencies:

```
DataFrames      Any Version
```

Contributors:

3.29 Distance

Current Version: `0.2.0`

Julia module for Distance evaluation

Maintainer: Dahua Lin

Dependencies:

```
Devectorize     Any Version
julia           [v"0.2.0-"]
```

Contributors:

3.30 Distributions

Current Version: `0.0.0`

A Julia package for probability distributions and associated funtions.

Maintainer: JuliaStats

Dependencies:

None

Contributors:

3.31 Elliptic

Current Version: `0.0.0`

Elliptic integral and Jacobi elliptic special functions

Maintainer: Mike Nolta

Dependencies:

None

Contributors:

3.32 Example

Current Version: `0.0.0`

Example Julia package repo.

Maintainer: The Julia Language

Dependencies:

None

Contributors:

3.33 FITSIO

Current Version: `0.0.0`

FITS file package for Julia

Maintainer: Mike Nolta

Dependencies:

None

Contributors:

3.34 FactCheck

Current Version: `0.0.0`

Midje-like testing for Julia

Maintainer: Zach Allaun

Dependencies:

None

Contributors:

3.35 FastaRead

Current Version: `0.2.0`

A fast FASTA reader for Julia

Maintainer: Carlo Baldassi

Dependencies:

```
GZip            Any Version
julia           [v"0.2.0-"]
```

Contributors:

3.36 FileFind

Current Version: `0.0.0`

File::Find implementation in Julia

Maintainer: John Myles White

Dependencies:

None

Contributors:

3.37 GLFW

Current Version: `0.0.0`

GLFW bindings for Julia. GLFW is a multi-platform library for opening a window, creating an OpenGL context, and managing input.

Maintainer: Jay Weisskopf

Documentation: http://www.glfw.org/

Dependencies:

None

Contributors:

3.38 GLM

Current Version: `0.0.0`

Generalized linear models in Julia

Maintainer: JuliaStats

Dependencies:

```
DataFrames     Any Version
Distributions  Any Version
```

Contributors:

3.39 GLPK

Current Version: `0.0.0`

GLPK wrapper module for Julia

Maintainer: Carlo Baldassi

Dependencies:

```
BinDeps          Any Version
```

Contributors:

3.40 GLUT

Current Version: `0.0.0`

Julia interface to GLUT

Maintainer: Robert Ennis

Dependencies:

```
GetC             Any Version
OpenGL           Any Version
```

Contributors:

3.41 GSL

Current Version: `0.0.0`

Julia interface to the GNU Scientific Library (GSL)

Maintainer: Jiahao Chen

Dependencies:

None

Contributors:

3.42 GZip

Current Version: `0.0.0`

A Julia interface for gzip functions in zlib

Maintainer: Kevin Squire

Documentation: https://gzipjl.readthedocs.org/en/latest/

Dependencies:

```
None
```

Contributors:

3.43 Gadfly

Current Version: `0.0.0`

Crafty statistical graphics for Julia.

Maintainer: Daniel Jones

Documentation: http://dcjones.github.com/Gadfly.jl/doc

Dependencies:

```
ArgParse       Any Version
Codecs         Any Version
Compose        Any Version
DataFrames     Any Version
Distributions  Any Version
Iterators      Any Version
JSON           Any Version
```

Contributors:

3.44 Gaston

Current Version: `0.0.0`

A julia front-end for gnuplot.

Maintainer: mbaz

Dependencies:

```
julia          [v"0.1.0-", v"0.2.0-"]
```

Contributors:

3.45 GetC

Current Version: `0.0.0`

Minimal implementation of Jasper's Julia FFI

Maintainer: Robert Ennis

Dependencies:

None

Contributors:

3.46 GoogleCharts

Current Version: `0.0.0`

Julia interface to Google Chart Tools

Maintainer: john verzani

Dependencies:

```
Calendar        Any Version
DataFrames      Any Version
JSON            Any Version
Mustache        Any Version
```

Contributors:

3.47 Graphs

Current Version: `0.0.0`

Working with graphs in Julia

Maintainer: John Myles White

Dependencies:

```
DataFrames      Any Version
```

Contributors:

3.48 Grid

Current Version: `0.2.0`

Interpolation and related operations on grids

Maintainer: Tim Holy

Dependencies:

None

Contributors:

3.49 Gtk

Current Version: `0.0.0`

Julia interface to Gtk windowing toolkit.

Maintainer: Jameson Nash

Dependencies:

`Cairo`	**Any** `Version`

Contributors:

3.50 Gurobi

Current Version: `0.0.0`

Julia Port of Gurobi Optimizer

Maintainer: Dahua Lin

Dependencies:

None

Contributors:

3.51 HDF5

Current Version: `0.2.0`

HDF5 interface for the Julia language

Maintainer: Tim Holy

Dependencies:

```
StrPack         Any Version
julia           [v"0.2.0-"]
```

Contributors:

3.52 HDFS

Current Version: `0.0.0`

A Julia to the Hadoop and Map-R filesystems

Maintainer: The Julia Language

Dependencies:

None

Contributors:

3.53 HTTP

Current Version: `0.0.2`

HTTP library (server, client, parser) for the Julia language

Maintainer: Dirk Gadsden

Dependencies:

```
Calendar        Any Version
```

Contributors:

3.54 Hadamard

Current Version: `0.0.0`

Fast Walsh-Hadamard transforms for the Julia language

Maintainer: Steven G. Johnson

Dependencies:

None

Contributors:

3.55 HypothesisTests

Current Version: `0.2.0`

T-tests, Wilcoxon rank sum (Mann-Whitney U), signed rank, and circular statistics in Julia

Maintainer: Simon Kornblith

Dependencies:

```
Distributions  Any Version
Rmath          Any Version
julia          [v"0.2.0-"]
```

Contributors:

3.56 ICU

Current Version: `0.0.0`

Julia wrapper for the International Components for Unicode (ICU) library

Maintainer: Mike Nolta

Dependencies:

```
UTF16          Any Version
```

Contributors:

3.57 Images

Current Version: `0.0.0`

An image library for Julia

Maintainer: Tim Holy

Dependencies:

None

Contributors:

3.58 ImmutableArrays

Current Version: `0.0.0`

Statically-sized immutable vectors and matrices.

Maintainer: Tracy Wadleigh

Dependencies:

```
julia           [v"0.2.0-"]
```

Contributors:

3.59 IniFile

Current Version: `0.0.0`

Reading and writing Windows-style INI files (writing not yet implemented).

Maintainer: The Julia Language

Dependencies:

None

Contributors:

3.60 Iterators

Current Version: `0.0.0`

Common functional iterator patterns.

Maintainer: The Julia Language

Dependencies:

None

Contributors:

3.61 Ito

Current Version: `0.0.0`

A Julia package for quantitative finance

Maintainer: Avik Sengupta

Documentation: http://aviks.github.com/Ito.jl/

Dependencies:

```
Calendar       Any Version
Distributions  Any Version
```

Contributors:

3.62 JSON

Current Version: `0.0.0`

JSON parsing and printing

Maintainer: Avik Sengupta

Dependencies:

None

Contributors:

3.63 JudyDicts

Current Version: `0.0.0`

Judy Array for Julia

Maintainer: Tanmay Mohapatra

Dependencies:

```
None
```

Contributors:

3.64 JuliaWebRepl

Current Version: `0.0.0`

Maintainer: Jameson Nash

Dependencies:

```
BinDeps         Any Version
julia           [v"0.2.0-"]
```

Contributors:

3.65 Jyacas

Current Version: `0.0.0`

Interface to use yacas from julia

Maintainer: john verzani

Dependencies:

```
JSON            Any Version
```

Contributors:

3.66 KLDivergence

Current Version: `0.0.0`

KL-divergence estimation in Julia

Maintainer: John Myles White

Dependencies:

```
Distributions  Any Version
```

Contributors:

3.67 LM

Current Version: `0.0.0`

Linear models in Julia

Maintainer: JuliaStats

Dependencies:

```
DataFrames     Any Version
Distributions  Any Version
```

Contributors:

3.68 Languages

Current Version: `0.0.0`

A package for working with human languages

Maintainer: John Myles White

Dependencies:

None

Contributors:

3.69 LazySequences

Current Version: `0.0.0`

Lazy sequences.

Maintainer: Daniel Jones

Dependencies:

None

Contributors:

3.70 LinProgGLPK

Current Version: `0.0.0`

High-level linear programming functionality for Julia via GLPK library (transitional package)

Maintainer: Carlo Baldassi

Dependencies:

```
GLPK            Any Version
```

Contributors:

3.71 Loss

Current Version: `0.0.0`

Loss functions

Maintainer: John Myles White

Dependencies:

None

Contributors:

3.72 MAT

Current Version: `0.2.0`

Julia module for reading MATLAB files

Maintainer: Simon Kornblith

Dependencies:

```
HDF5            Any Version
julia           [v"0.2.0-"]
```

Contributors:

3.73 MATLAB

Current Version: `0.0.0`

Calling MATLAB in Julia through MATLAB Engine

Maintainer: Dahua Lin

Dependencies:

None

Contributors:

3.74 MCMC

Current Version: `0.0.0`

MCMC tools for Julia

Maintainer: Chris DuBois

Dependencies:

```
Options         Any Version
```

Contributors:

3.75 MLBase

Current Version: `0.0.0`

A set of functions to support the development of machine learning algorithms

Maintainer: Dahua Lin

Dependencies:

```
Devectorize     Any Version
Distance        Any Version
```

Contributors:

3.76 MarketTechnicals

Current Version: `0.0.0`

Technical analysis of financial time series in Julia

Maintainer: milktrader

Dependencies:

```
Calendar        Any Version
DataFrames      Any Version
Stats           Any Version
TimeSeries      Any Version
UTF16           Any Version
julia           [v"0.1.0-", v"0.2.0-"]
```

Contributors:

3.77 MathProg

Current Version: `0.0.0`

Modelling language for Linear, Integer, and Quadratic Programming

Maintainer: Iain Dunning

Dependencies:

```
Clp             Any Version
CoinMP          Any Version
julia           [v"0.2.0-"]
```

Contributors:

3.78 MathProgBase

Current Version: `0.0.0`

Provides standard interface to linear programming solvers, including linprog function.

Maintainer: Miles Lubin

Dependencies:

```
julia           [v"0.1.0-"]
```

Contributors:

3.79 Meshes

Current Version: `0.0.0`

Generation and manipulation of triangular meshes.

Maintainer: Tracy Wadleigh

Dependencies:

None

Contributors:

3.80 MixedModels

Current Version: `0.0.0`

A Julia package for fitting (statistical) mixed-effects models

Maintainer: dmbates

Dependencies:

```
Distributions  Any Version
NLopt          Any Version
julia          [v"0.2.0-"]
```

Contributors:

3.81 Monads

Current Version: `0.0.0`

Monadic expressions and sequences for Julia

Maintainer: Patrick O'Leary

Documentation: https://monadsjl.readthedocs.org/

Dependencies:

None

Contributors:

3.82 Mongo

Current Version: `0.0.0`

Mongo bindings for the Julia programming language

Maintainer: Brian Smith

Dependencies:

None

Contributors:

3.83 Mongrel2

Current Version: `0.0.0`

Mongrel2 handlers in Julia

Maintainer: Avik Sengupta

Dependencies:

```
JSON            Any Version
ZMQ             Any Version
```

Contributors:

3.84 Mustache

Current Version: `0.0.0`

Port of mustache.js to julia

Maintainer: john verzani

Dependencies:

```
DataFrames      Any Version
```

Contributors:

3.85 NHST

Current Version: `0.0.0`

Null hypothesis significance tests

Maintainer: John Myles White

Dependencies:

```
None
```

Contributors:

3.86 NLopt

Current Version: `0.0.0`

Package to call the NLopt nonlinear-optimization library from the Julia language

Maintainer: Steven G. Johnson

Dependencies:

```
julia           [v"0.2.0-"]
```

Contributors:

3.87 Named

Current Version: `0.0.0`

Julia named index and named vector types

Maintainer: Harlan Harris

Dependencies:

None

Contributors:

3.88 ODBC

Current Version: `0.0.0`

A low-level ODBC interface for the Julia programming language

Maintainer: Jacob Quinn

Dependencies:

`DataFrames`	**Any** Version

Contributors:

3.89 ODE

Current Version: `0.0.0`

Assorted basic Ordinary Differential Equation solvers

Maintainer: Jameson Nash

Dependencies:

`Polynomial`	**Any** Version

Contributors:

3.90 OpenGL

Current Version: `0.0.0`

Julia interface to OpenGL

Maintainer: Robert Ennis

Dependencies:

```
GetC            Any Version
```

Contributors:

3.91 OpenSSL

Current Version: `0.0.0`

WIP OpenSSL bindings for Julia

Maintainer: Dirk Gadsden

Dependencies:

None

Contributors:

3.92 Optim

Current Version: `0.0.0`

Optimization functions for Julia

Maintainer: John Myles White

Documentation: http://johnmyleswhite.com

Dependencies:

```
Calculus        Any Version
Distributions   Any Version
Options         Any Version
```

Contributors:

3.93 Options

Current Version: `0.2.0`

A framework for providing optional arguments to functions.

Maintainer: The Julia Language

Dependencies:

```
julia           [v"0.2.0-"]
```

Contributors:

3.94 PLX

Current Version: `0.0.0`

Julia module for reading Plexon PLX files

Maintainer: Simon Kornblith

Dependencies:

None

Contributors:

3.95 PatternDispatch

Current Version: `0.0.0`

Method dispatch based on pattern matching for Julia

Maintainer: toivoh

Dependencies:

None

Contributors:

3.96 Polynomial

Current Version: `0.0.0`

Polynomial manipulations

Maintainer: Jameson Nash

Dependencies:

None

Contributors:

3.97 Profile

Current Version: `0.2.0`

Profilers for Julia

Maintainer: Tim Holy

Dependencies:

```
julia           [v"0.2.0-"]
```

Contributors:

3.98 ProjectTemplate

Current Version: `0.0.0`

ProjectTemplate for Julia

Maintainer: John Myles White

Dependencies:

```
DataFrames      Any Version
JSON            Any Version
```

Contributors:

3.99 PyCall

Current Version: `0.0.0`

Package to call Python functions from the Julia language

Maintainer: Steven G. Johnson

Dependencies:

```
julia           [v"0.2.0-"]
```

Contributors:

3.100 QuickCheck

Current Version: `0.0.0`

QuickCheck specification-based testing for Julia

Maintainer: Patrick O'Leary

Documentation: https://quickcheckjl.readthedocs.org/

Dependencies:

```
None
```

Contributors:

3.101 RDatasets

Current Version: `0.0.0`

Julia package for loading many of the data sets available in R

Maintainer: John Myles White

Dependencies:

```
DataFrames      Any Version
```

Contributors:

3.102 RNGTest

Current Version: `0.0.0`

Code for testing of Julia's random numbers

Maintainer: Andreas Noack Jensen

Dependencies:

None

Contributors:

3.103 RandomMatrices

Current Version: `0.0.0`

Random matrices package for Julia

Maintainer: Jiahao Chen

Dependencies:

```
Catalan        Any Version
Distributions  Any Version
GSL            Any Version
ODE            Any Version
```

Contributors:

3.104 Resampling

Current Version: `0.0.0`

Tools for resampling data in Julia

Maintainer: John Myles White

Dependencies:

```
DataFrames     Any Version
```

Contributors:

3.105 Rif

Current Version: `0.0.0`

Julia-to-R interface

Maintainer: Laurent Gautier

Dependencies:

None

Contributors:

3.106 Rmath

Current Version: `0.0.0`

Archive of functions that emulate R's d-p-q-r functions for probability distributions

Maintainer: dmbates

Dependencies:

None

Contributors:

3.107 SDE

Current Version: `0.0.0`

Simulation and inference for Ito processes and diffusions.

Maintainer: M. Schauer

Dependencies:

None

Contributors:

3.108 SDL

Current Version: `0.0.0`

Julia interface to SDL

Maintainer: Robert Ennis

Dependencies:

```
GetC            Any Version
OpenGL          Any Version
```

Contributors:

3.109 SemidefiniteProgramming

Current Version: `0.0.0`

Interface to semidefinite programming libraries.

Maintainer: David de Laat

Dependencies:

None

Contributors:

3.110 SimJulia

Current Version: `0.0.0`

Process oriented simulation library written in Julia

Maintainer: Ben Lauwens

Dependencies:

None

Contributors:

3.111 Sims

Current Version: `0.0.0`

Experiments with non-causal, equation-based modeling in Julia

Maintainer: Tom Short

Dependencies:

None

Contributors:

3.112 Stats

Current Version: `0.2.0`

Basic statistics for Julia

Maintainer: JuliaStats

Dependencies:

```
julia           [v"0.2.0-"]
```

Contributors:

3.113 StrPack

Current Version: `0.0.0`

Swiss Army Knife for encoding and decoding binary streams

Maintainer: Patrick O'Leary

Documentation: https://strpackjl.readthedocs.org/

Dependencies:

```
julia           [v"0.2.0-"]
```

Contributors:

3.114 Sundials

Current Version: `0.0.0`

Julia interface to Sundials, including a nonlinear solver (KINSOL), ODE's (CVODE), and DAE's (IDA).

Maintainer: Tom Short

Dependencies:

```
julia           [v"0.2.0-"]
```

Contributors:

3.115 SymbolicLP

Current Version: `0.0.0`

Symbolic linear programming and linear constraints

Maintainer: Tim Holy

Dependencies:

None

Contributors:

3.116 TOML

Current Version: `0.0.0`

A TOML parser for Julia.

Maintainer: pygy

Dependencies:

```
Calendar        Any Version
JSON            Any Version
```

Contributors:

3.117 TextAnalysis

Current Version: `0.0.0`

Julia package for text analysis

Maintainer: John Myles White

Dependencies:

```
DataFrames      Any Version
Languages       Any Version
```

Contributors:

3.118 TextWrap

Current Version: `0.0.0`

Package for wrapping text into paragraphs.

Maintainer: Carlo Baldassi

Dependencies:

```
Options         Any Version
```

Contributors:

3.119 TimeModels

Current Version: `0.0.0`

Modeling time series in Julia

Maintainer: milktrader

Dependencies:

```
Calendar        Any Version
DataFrames      Any Version
Stats           Any Version
TimeSeries      Any Version
UTF16           Any Version
julia           [v"0.1.0-", v"0.2.0-"]
```

Contributors:

3.120 TimeSeries

Current Version: `0.0.0`

Time series toolkit for Julia

Maintainer: milktrader

Dependencies:

```
Calendar        Any Version
DataFrames      Any Version
Stats           Any Version
UTF16           Any Version
julia           [v"0.1.0-", v"0.2.0-"]
```

Contributors:

3.121 Tk

Current Version: `0.0.0`

Julia interface to Tk windowing toolkit.

Maintainer: The Julia Language

Dependencies:

```
BinDeps         Any Version
Cairo           Any Version
```

Contributors:

3.122 TkExtras

Current Version: `0.0.0`

Additions to the Tk.jl pacakge

Maintainer: john verzani

Dependencies:

```
Tk              Any Version
```

Contributors:

3.123 TopicModels

Current Version: `0.0.0`

TopicModels for Julia

Maintainer: Jonathan Chang

Dependencies:

None

Contributors:

3.124 TradingInstrument

Current Version: `0.0.0`

Downloading financial time series data and providing financial asset types in Julia

Maintainer: milktrader

Dependencies:

```
Calendar       Any Version
DataFrames     Any Version
Stats          Any Version
TimeSeries     Any Version
UTF16          Any Version
julia          [v"0.1.0-", v"0.2.0-"]
```

Contributors:

3.125 Trie

Current Version: `0.0.0`

Implementation of the trie data structure.

Maintainer: The Julia Language

Dependencies:

None

Contributors:

3.126 UTF16

Current Version: `0.0.0`

UTF16 string type for Julia

Maintainer: Mike Nolta

Dependencies:

None

Contributors:

3.127 Units

Current Version: `0.0.0`

Infrastructure for handling physical units for the Julia programming language

Maintainer: Tim Holy

Dependencies:

None

Contributors:

3.128 WAV

Current Version: `0.1.0`

Julia package for working with WAV files

Maintainer: Daniel Casimiro

Dependencies:

```
Options         Any Version
julia           [v"0.1.0-", v"0.2.0-"]
```

Contributors:

3.129 Winston

Current Version: `0.0.0`

2D plotting for Julia

Maintainer: Mike Nolta

Dependencies:

```
Cairo           Any Version
Color           Any Version
IniFile         Any Version
Tk              Any Version
```

Contributors:

3.130 ZMQ

Current Version: `0.0.0`

Julia interface to ZMQ

Maintainer: Avik Sengupta

Dependencies:

None

Contributors:

3.131 Zlib

Current Version: `0.0.0`

zlib bindings for Julia

Maintainer: Daniel Jones

Dependencies:

None

Contributors:

3.132 kNN

Current Version: `0.0.0`

The k-nearest neighbors algorithm in Julia

Maintainer: John Myles White

Dependencies:

`DataFrames`	**`Any`** `Version`

Contributors:

b

b