Julia

Not a simple language.

Strengths

- performance
- parallelism out of the box
- scientific computing (a *lot* of amenities)

Typing

- “dynamically”-typed
- optionally typed with type inference

Overview

- `x = 10`
- meta programming (macros)
- functional programming: `id = a -> a`
- bang`!` convention
- usual PL features (loops, functions, etc.)

The Amenities

- unicode
- `Int8`, `UInt8`, ..., `Int128`, `UInt128`, `Float16` (32 & 64)
- `1 + 2im` (complex numbers)
- `-4//-12 == 1//3` (rational numbers)
- binary (`0b10`), octal (`0o010`), hexa (`0xA`) values
- chaining comparisons: `0 <= a < 1`
- *much* more…

```julia
# Σ(\Sigma)

# nice and readable
function Σ1(f, n::Int, m::Int)::Int
    sum = 0
    for i = n:m
        sum += f(i)
    end
    return sum
end

n = 1
m = 3
f = a -> a^2
result = Σ1(f, n, m)

print("Normal (Σ1): ")
println(result)


# weird and compact
Σ2(f, n, m) = n ≤ m ? f(n) + Σ2(f, n + 1, m) : 0

arguments = (f, n, m)
result = Σ2(arguments...)

print("Recursive (Σ2): ")
println(result)

```

Parallelism

- Julia Coroutines (Green Threading)
- Multi-Threading (Experimental)
- Multi-Core or Distributed Processing

Coroutines

```julia
# tasks (symmetric coroutines)
println("-- Tasks")

producer = @task begin
    for i = 1:3
        yieldto(consumer, i * 2)
    end
end

consumer = @task begin
    while true
        println(yieldto(producer))
    end
end

schedule(producer)
schedule(consumer)
wait(producer)
```

```julia
# channels
println("-- Channels")

function producer(c)
    for i = 1:5
        put!(c, i * 2)
    end
    close(c)
end

c = Channel{Int}(1)
@async producer(c)

for i in c
   println(i)
end
```

Other Task Functions

- **`yield():`** yields to the scheduler. A task that calls this function is still runnable, and will be restarted immediately if there are no other runnable tasks.
- **`wait(task):`** blocks until the given task is done. (wait behaves differently depending on its argument's type.)
- **`fetch(channel):`** waits until there is an item in the channel and returns the item without removing it.
- **`notify(condition, ...):`** wake up tasks waiting for a condition.
- **`bind(channel, task):`** associates the lifetime of channel with a task.

Multi-Threading

- likely to change
- starts up with JULIA_NUM_THREADS (default 1)
- synchronization primitives (locks, semaphores, etc) are available
- functions `Threads.nthreads` and `Threads.threadid`
- ```
  @threads for i = 1:10
      array[i] = Threads.threadid()
  end
  ```
- atomicity:
  ```
  a = Threads.Atomic{Int}(15)
  Threads.atomic_add!(a, 5)
  ```
  ```

Distributed Processing

- `julia -p n` (starts with *n* local worker processes) or use the `--machine-file` option to start Julia processes on other machines

- `Distributed` module within the standard library

- message passing through remote calls

- remote references (`Future` and `RemoteChannel`)

Distributed Processing

- remote calls return a `Future` (immediately)

  ```
  sum = remotecall(+, myid(), 2, 3)
  fetch(sum) # 5
  # same as remotecall_fetch(+, myid(), 2, 3)
  ```

- `remotecall` is low-level, best to use `@spawn` most of the time

  ```
  sum = @spawn 2 + 3
  fetch(sum)
  ```

- `@async` is similar to `@spawn`, but only runs tasks on the local process

- a normal `Channel` can't be shared between workers, hence the `RemoteChannel`

Distributed Processing

- `@distributed` (reduce)
  ```
  @distributed (+) for i = 1:10 i end
  # result is 55
  ```

- without the reduce function, `@distributed` executes asynchronously and returns an array of futures immediately without waiting completion

- `@everywhere` (all processes)
  ```
  @everywhere include("SuperDuperModule.jl")
  # loads the module on all processes
  ```

- `@sync` synchronises enclosed `@async`, `@spawn`, `@spawnat` `@distributed` macros

- `pmap` function, just like `map`, but parallel

Distributed Processing

- MPI

  `import MPI`

- a Julia wrapper of the MPI protocol

- `mpirun -np 4 ./julia example.jl`

parallel depth-first scheduling

?