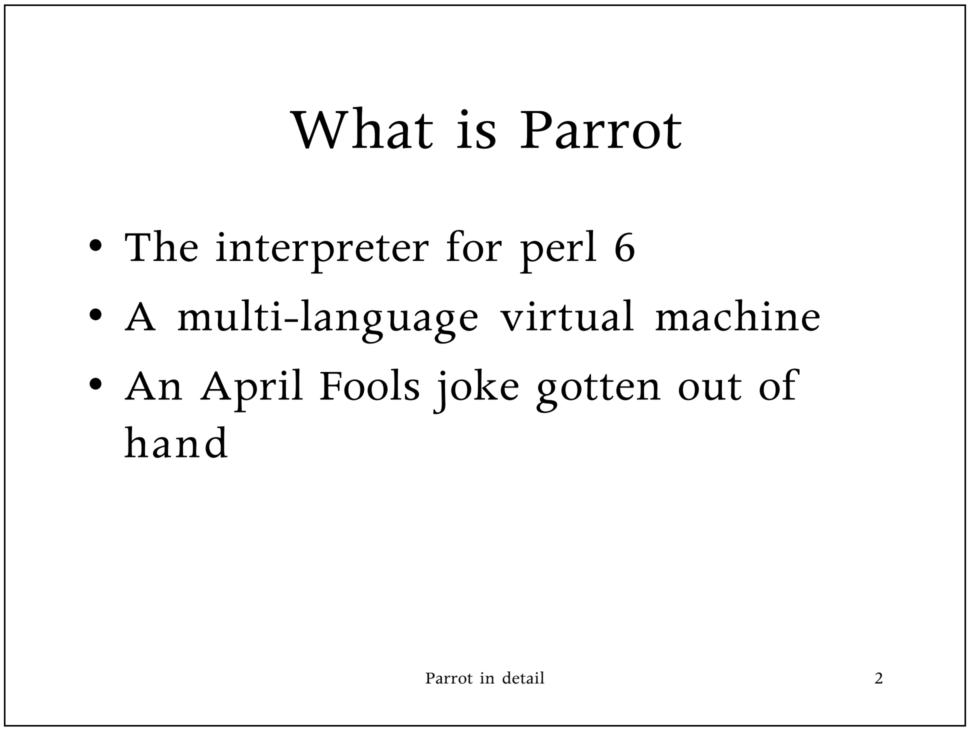
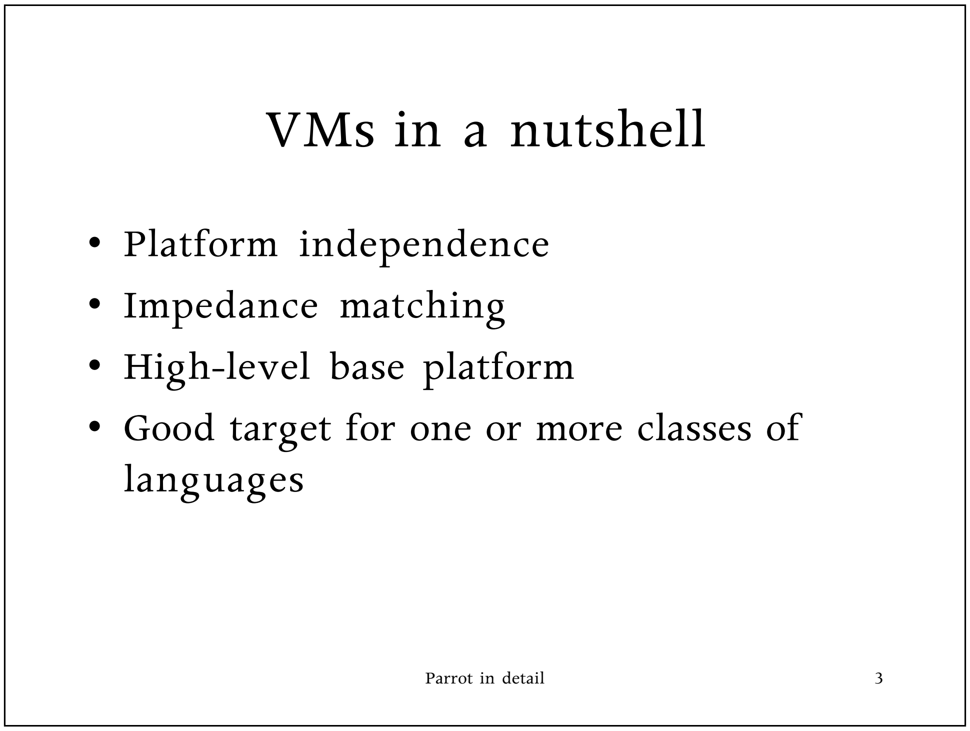
Parrot in Detail



What is Parrot

- The interpreter for perl 6
- A multi-language virtual machine
- An April Fools joke gotten out of hand

The joke was entirely Simon Cozens' fault^Wresponsibility

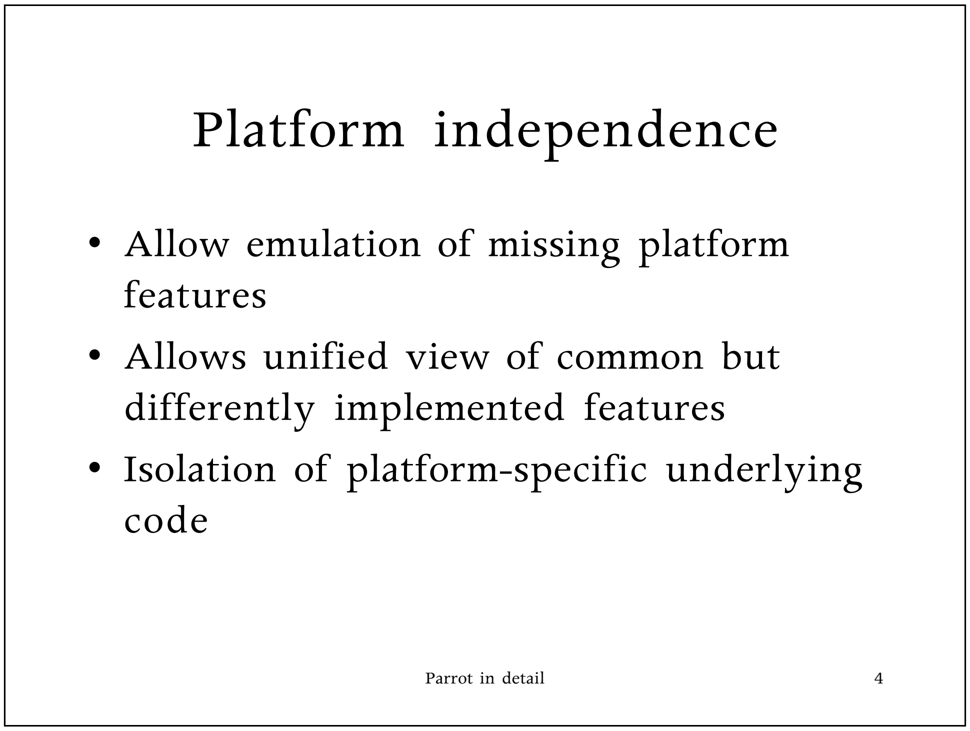


VMs in a nutshell

- Platform independence
- Impedance matching
- High-level base platform
- Good target for one or more classes of languages

Parrot in detail 3

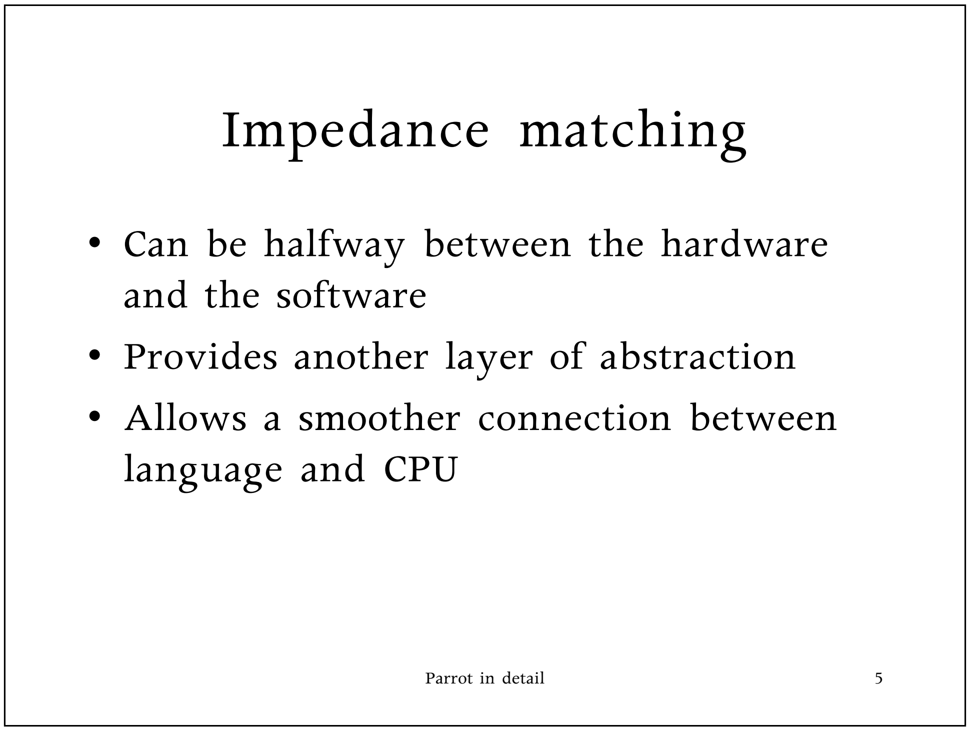
VM == fake hardware. This isn’t anything new, we’ve been doing it for decades.



Platform independence

- Allow emulation of missing platform features
- Allows unified view of common but differently implemented features
- Isolation of platform-specific underlying code

Like async I/O, threads, event handling, large files…



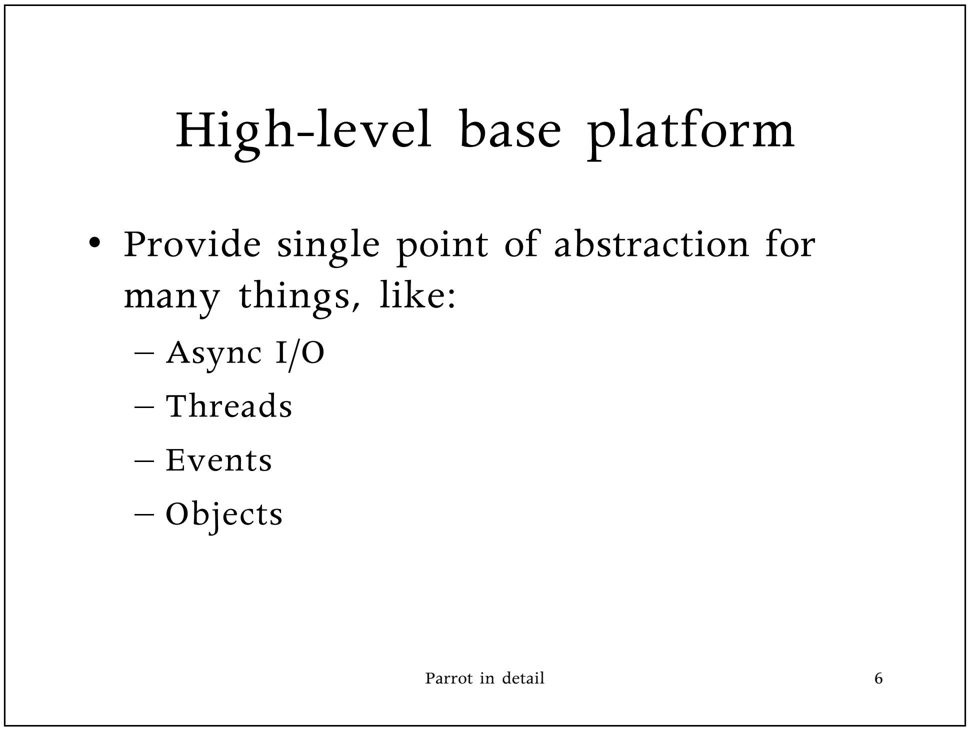
Impedance matching

- Can be halfway between the hardware and the software
- Provides another layer of abstraction
- Allows a smoother connection between language and CPU

Parrot in detail 5

The closer your implementation platform is to what you're implementing, the easier it is and the less 'friction' there is between layers. Sometimes adding a thunking layer is enough to make things a lot easier.

This is not unlike electronics, where you really want to match the impedence of circuitry.



High-level base platform

- Provide single point of abstraction for many things, like:
  - Async I/O
  - Threads
  - Events
  - Objects

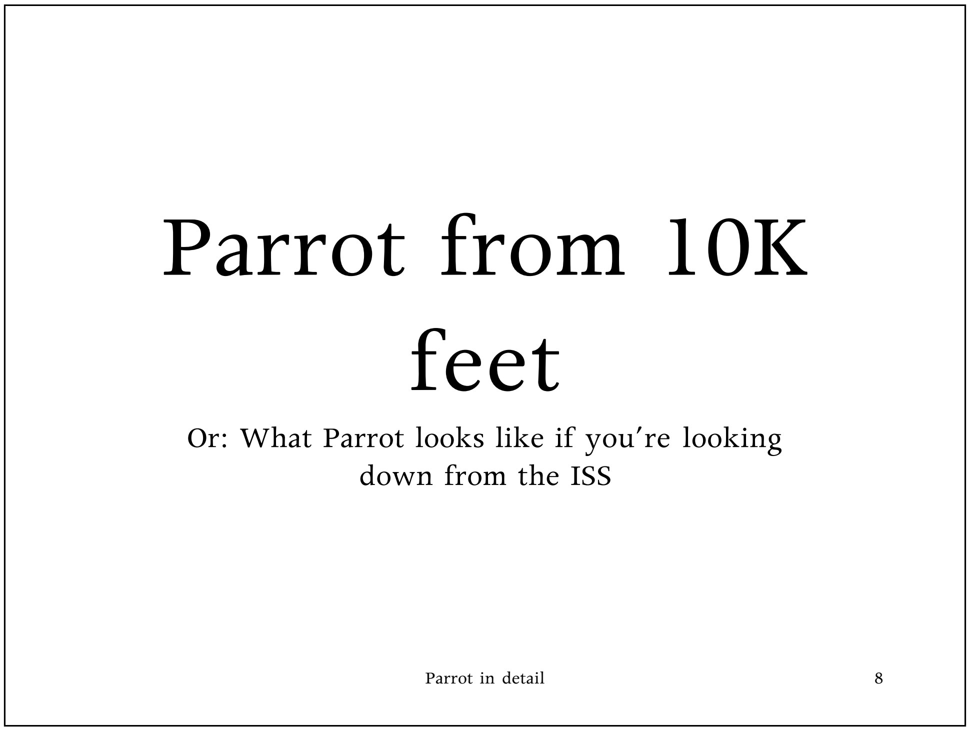
Parrot in detail 6

It’s a lot easier to present a single known interface and have each platform twiddle the bits they need to to implement that interface. Threads and async I/O are two big examples of that.

Building events and objects into the underlying platform (at least as far as the languages that use it are concerned) make language implementation easier and facilitates interoperability between routines in different languages.

Good HLL target

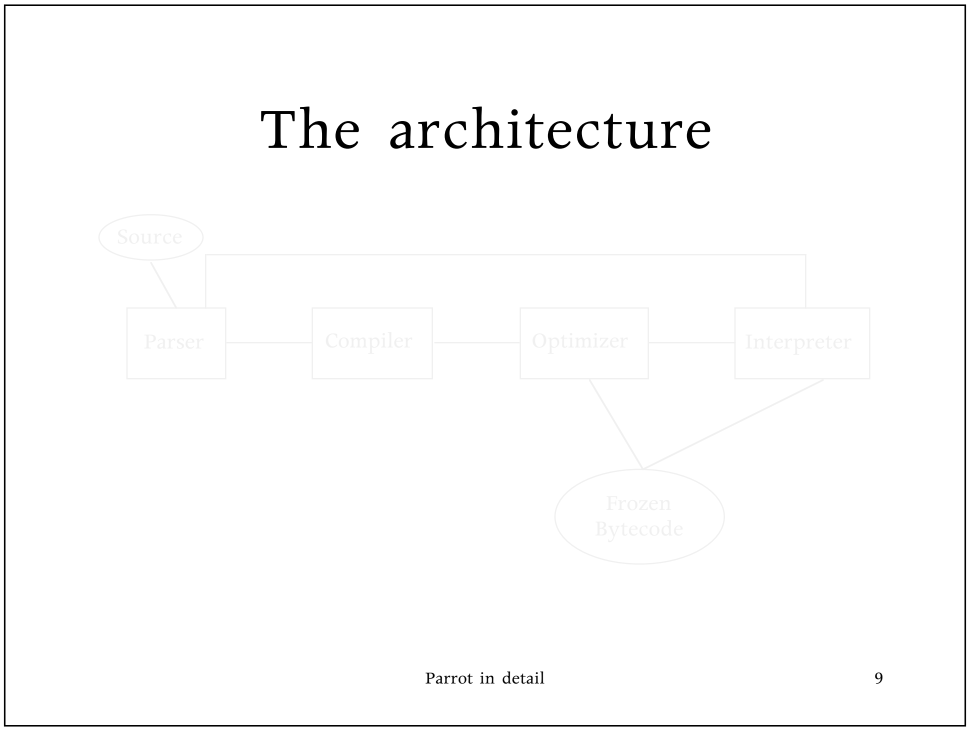
- Provide features that map well to a class of language constructs
- Reduce the “thought” load for compiler writers
- Allow tasks to be better partitioned and placed



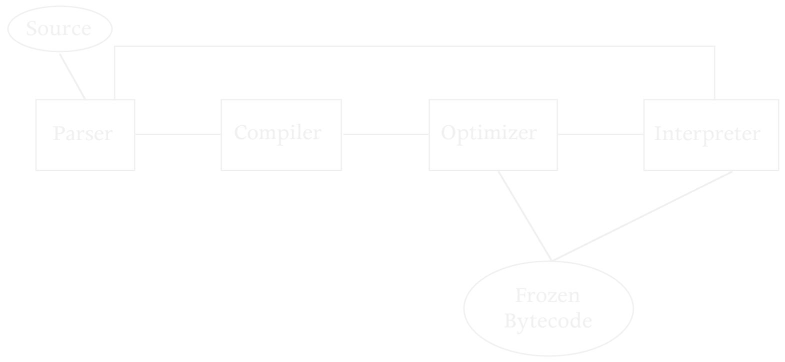
Parrot from 10K feet

Or: What Parrot looks like if you're looking down from the ISS

ISS is the international space station, for those who don't know. (It ought to be 10K kilometers, really. Give or take a bit)



The architecture

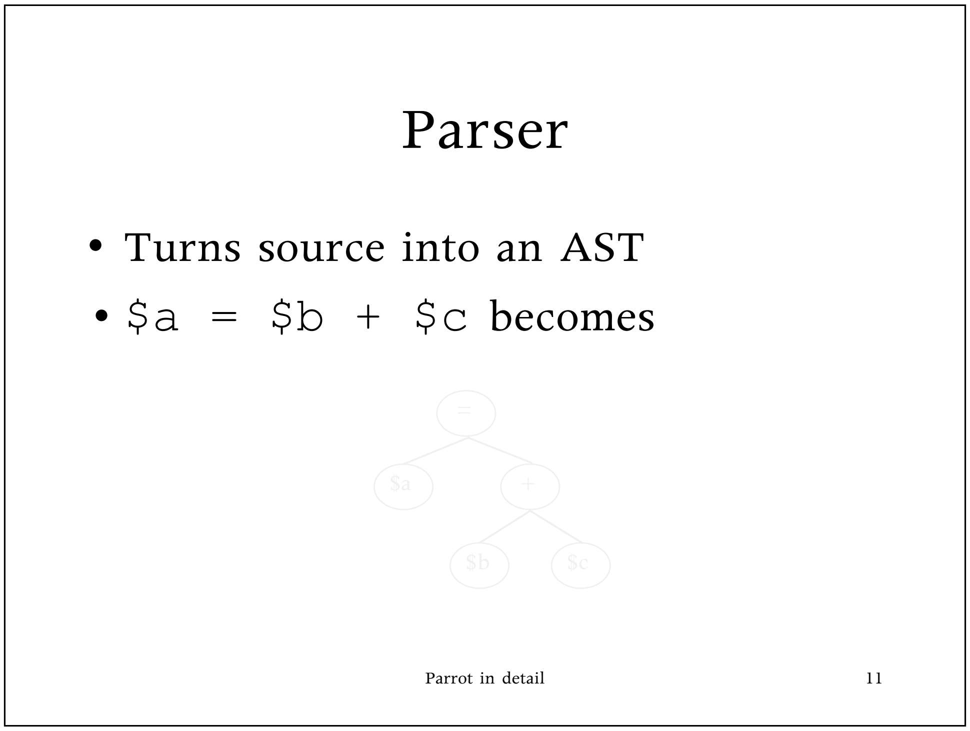


Parrot in detail 9

AppleWorks doesn't have arrows, but the structure's still reasonably clear. The connection to frozen bytecode's two ways--the optimizer and interpreter can either freeze it or load it

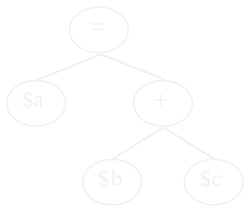
Parts and pieces

- Parser
- Compiler
- Optimizer
- Interpreter



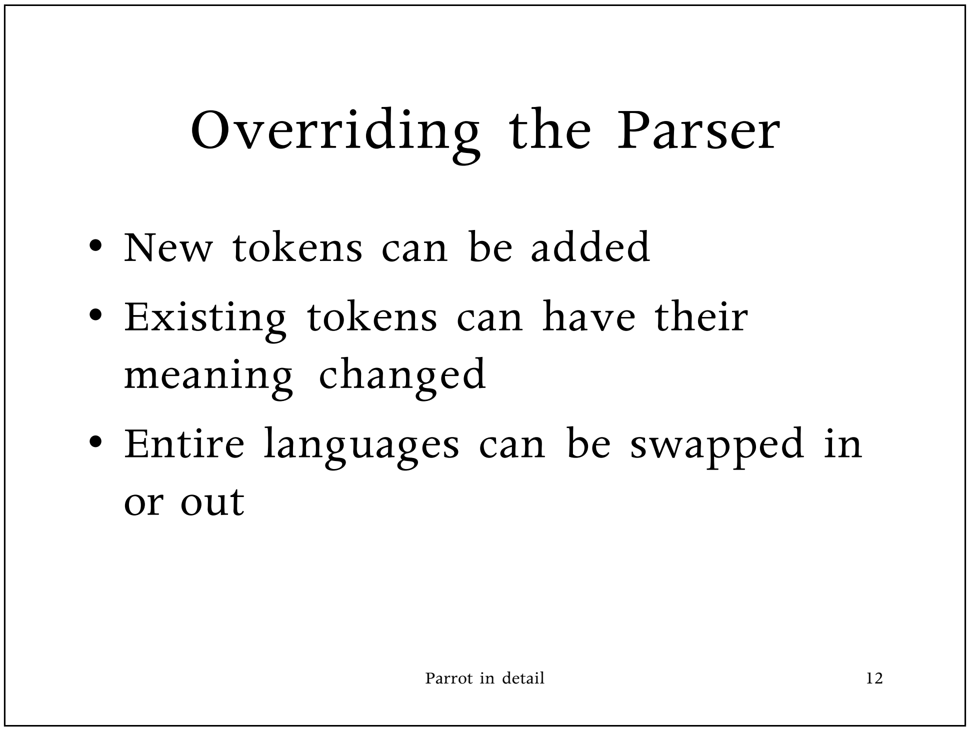
Parser

- Turns source into an AST
- `$a = $b + $c` becomes



Parrot in detail 11

This is a combination of tokenizing and parsing, but parrot's regex engine (courtesy of the steroids recently introduced into Perl's regexes) can act as a parser as well as just a regex engine.



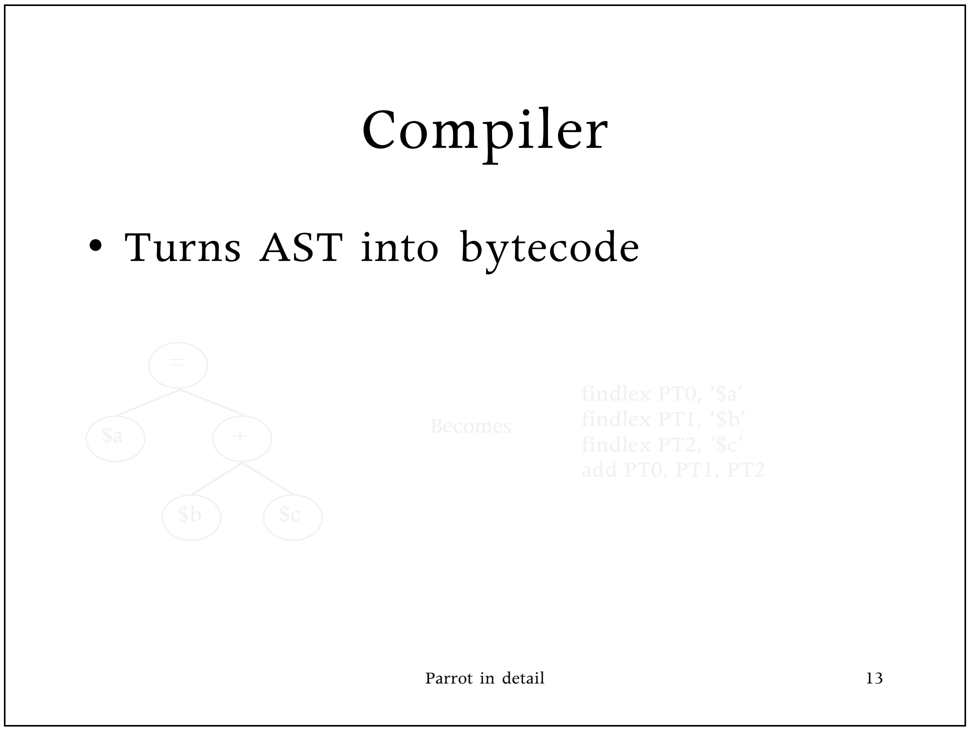
Overriding the Parser

- New tokens can be added
- Existing tokens can have their meaning changed
- Entire languages can be swapped in or out

Parrot in detail 12

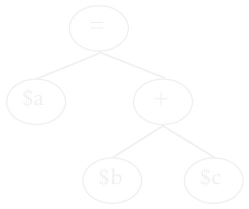
Makes swapping in new languages on the fly a lot easier. Also makes temporarily enhancing or overrriding parts of an existing language easier.

This is how lexically-overridden operators will be done--if + should be - for *all* operations in a block, this is where you'd do it. (If you're overriding only for a single class, that'd be done elsewhere, in the class vtable)



Compiler

- Turns AST into bytecode

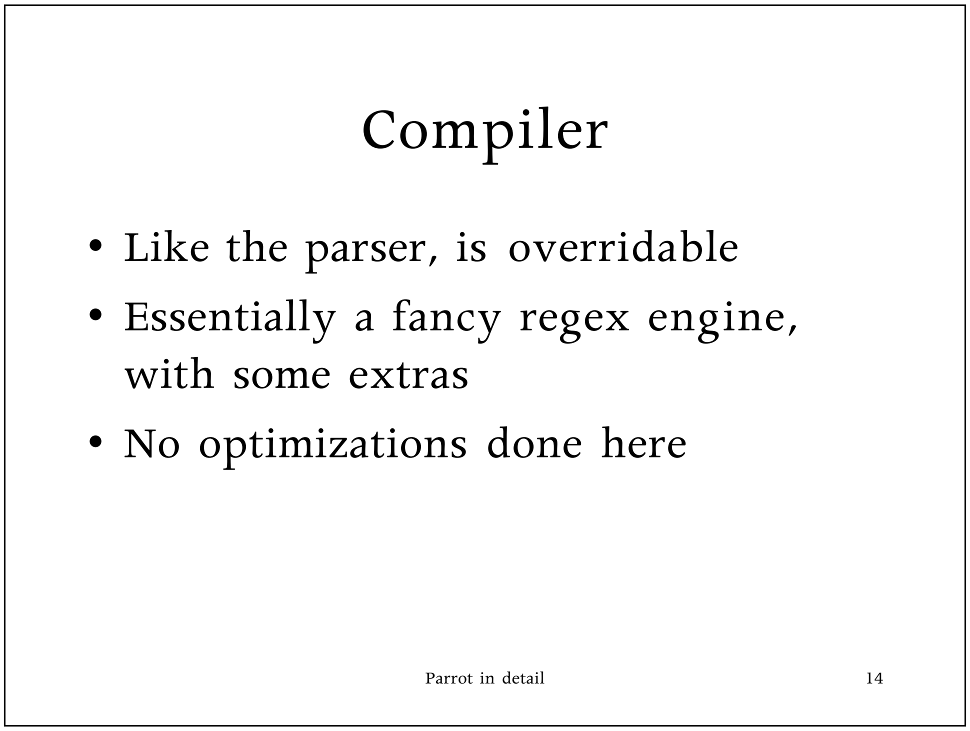


Becomes

```
findlex PT0, '$a'
findlex PT1, '$b'
findlex PT2, '$c'
add PT0, PT1, PT2
```

Parrot in detail 13

The PTx entries are temp PMC locations that the register coloring algorithm will later allocate to real PMC registers. (And we do have a register coloring algorithm now)

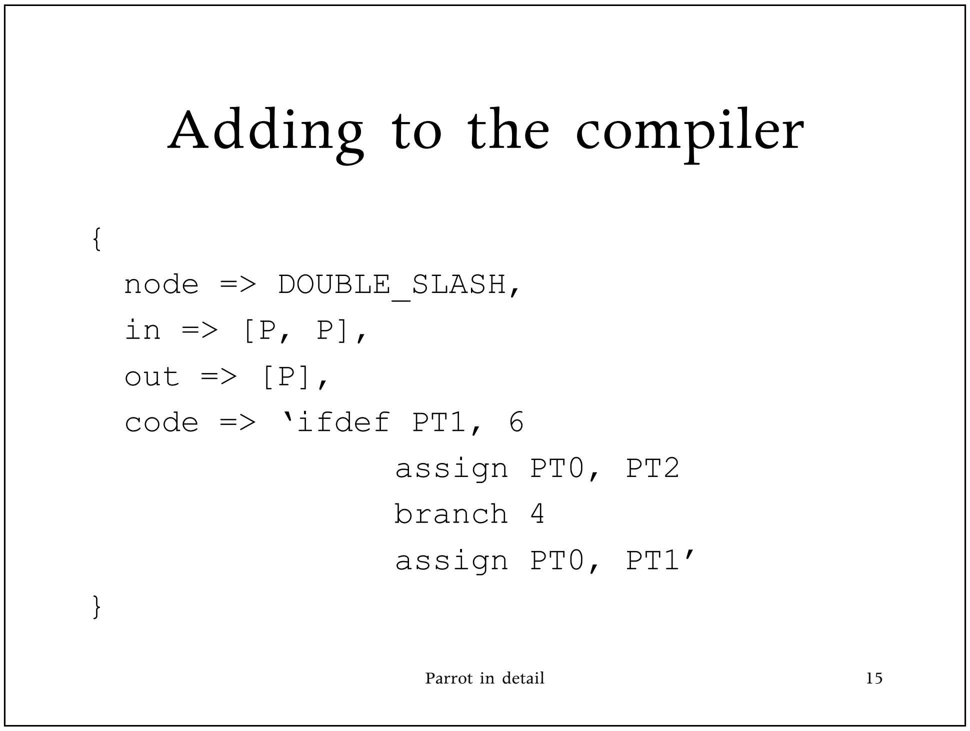


Compiler

- Like the parser, is overridable
- Essentially a fancy regex engine, with some extras
- No optimizations done here

Parrot in detail 14

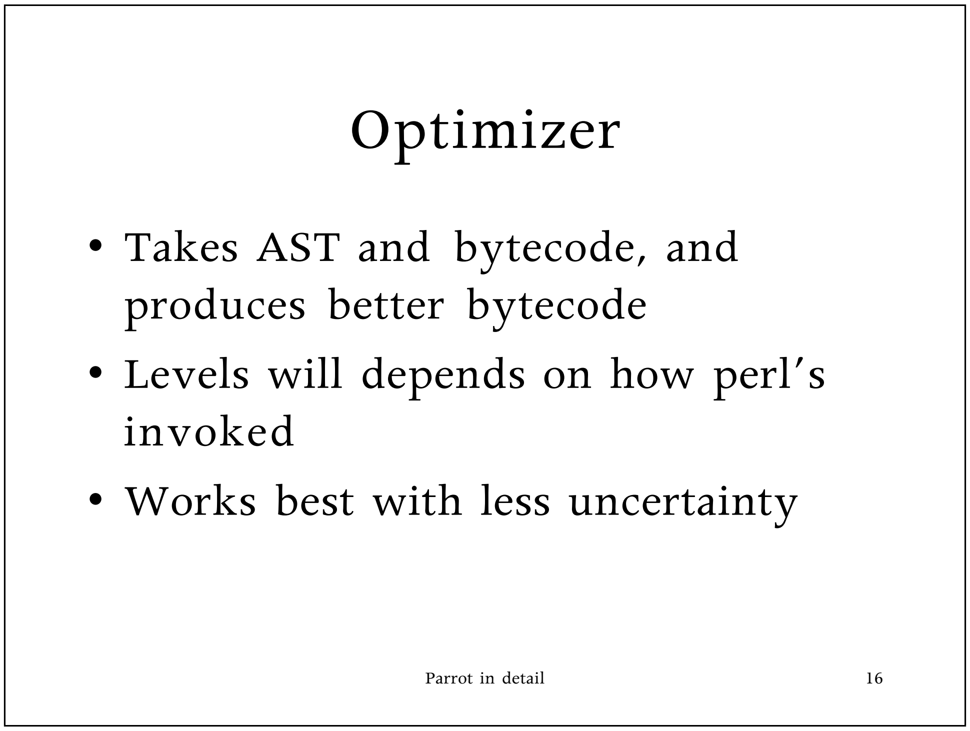
This is the place that the tree that the parser built up is transformed into bytecode. Definitely very well traveled ground.



Adding to the compiler

```
{
  node => DOUBLE_SLASH,
  in => [P, P],
  out => [P],
  code => ‘ifdef PT1, 6
                assign PT0, PT2
                branch 4
                assign PT0, PT1’
}
```

It’ll probably look different than this does, this is just an example to give a feel for what you might do.

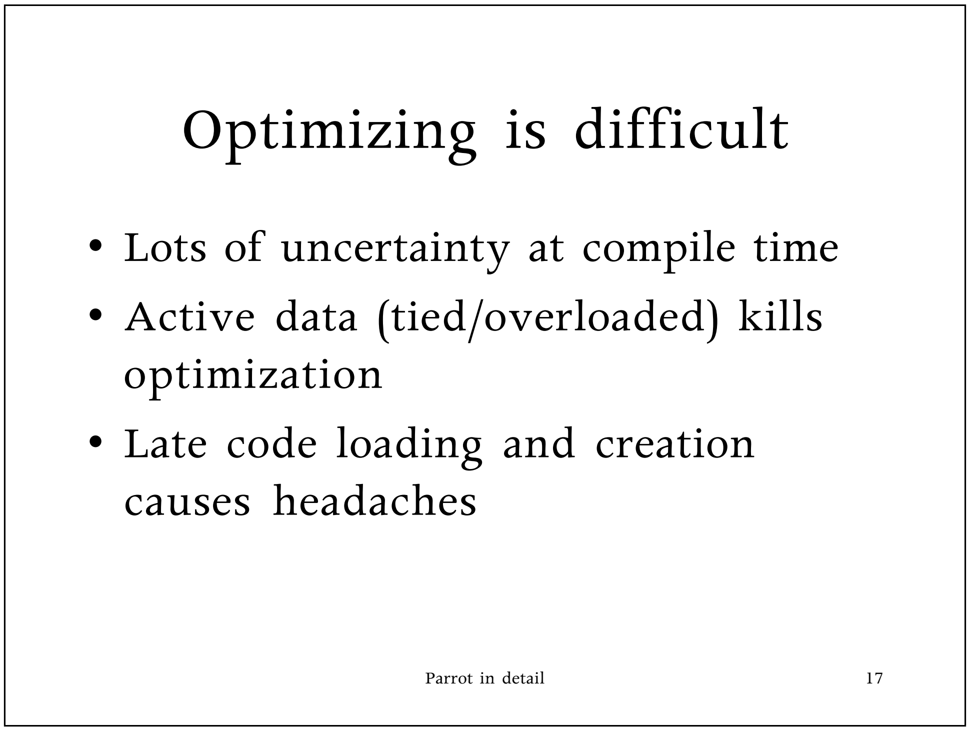


Optimizer

- Takes AST and bytecode, and produces better bytecode
- Levels will depends on how perl's invoked
- Works best with less uncertainty

Parrot in detail 16

Optimizing is… interesting. Alas, many of the languages that Parrot is well suited for are horrid to optimize, as we'll see.

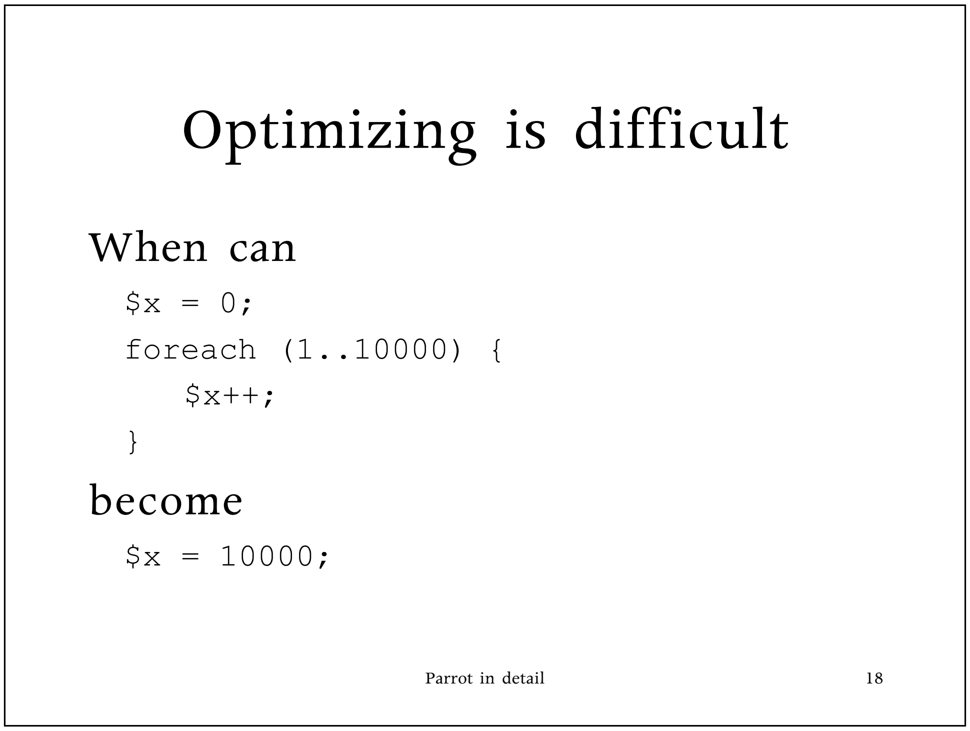


Optimizing is difficult

- Lots of uncertainty at compile time
- Active data (tied/overloaded) kills optimization
- Late code loading and creation causes headaches

Perl, generally, is almost impossible to optimize. (But you already knew that) I generally make it a point to note that, while as a language *implementor* it's a pain, as a language *user* I really like the features that make it tough to optimize.

Optimization (really de-pessimization, but who's quibbling?) is really cheating, and we can only cheat when we know nobody can look. That's not too often, alas.



Optimizing is difficult

When can

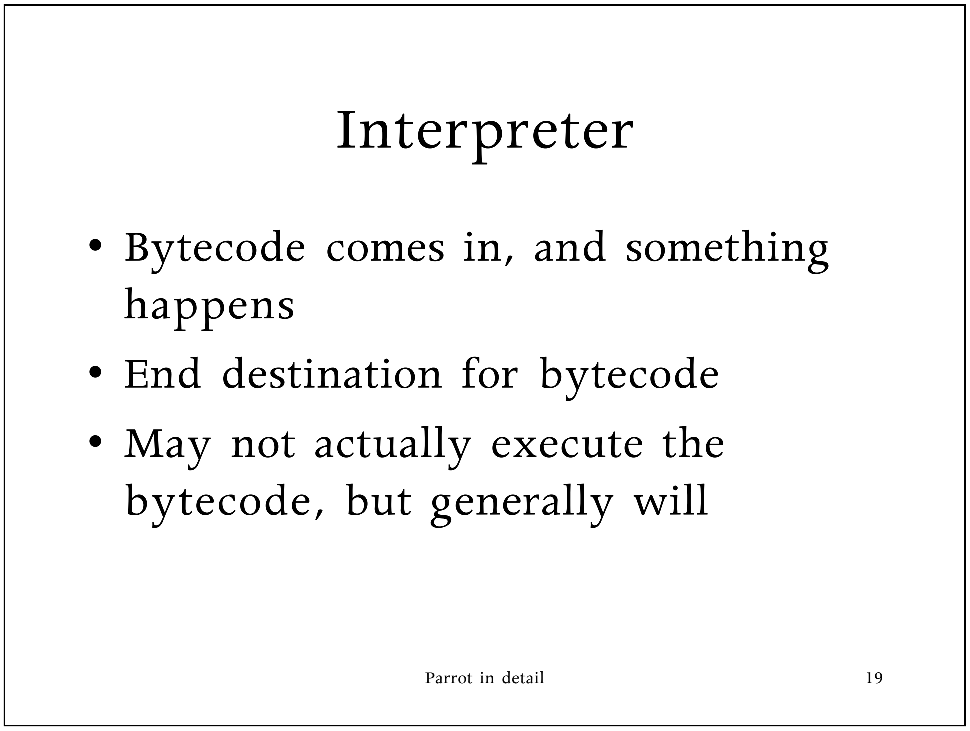
```
$x = 0;
foreach (1..10000) {
   $x++;
}
```

become

```
$x = 10000;
```

Parrot in detail 18

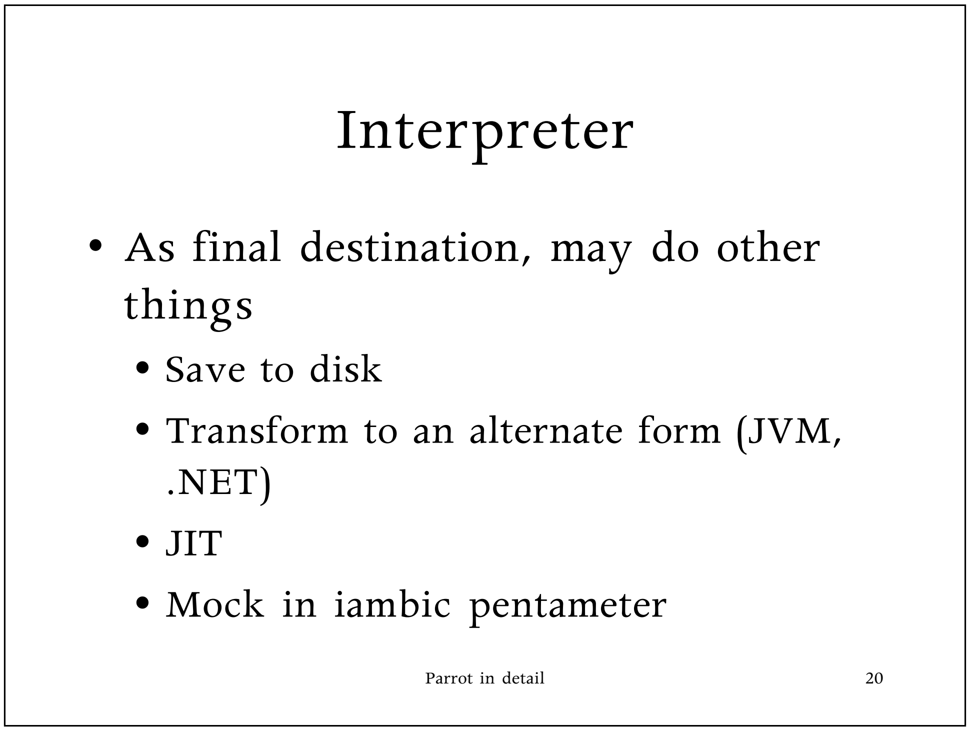
And the answer's potentially never, of course. If $x is tied, overloaded, or otherwise active we can't do this.



Interpreter

- Bytecode comes in, and something happens
- End destination for bytecode
- May not actually execute the bytecode, but generally will

I make this point because the interpreter part of the picture may not interpret anything. The next slide points out what might happen

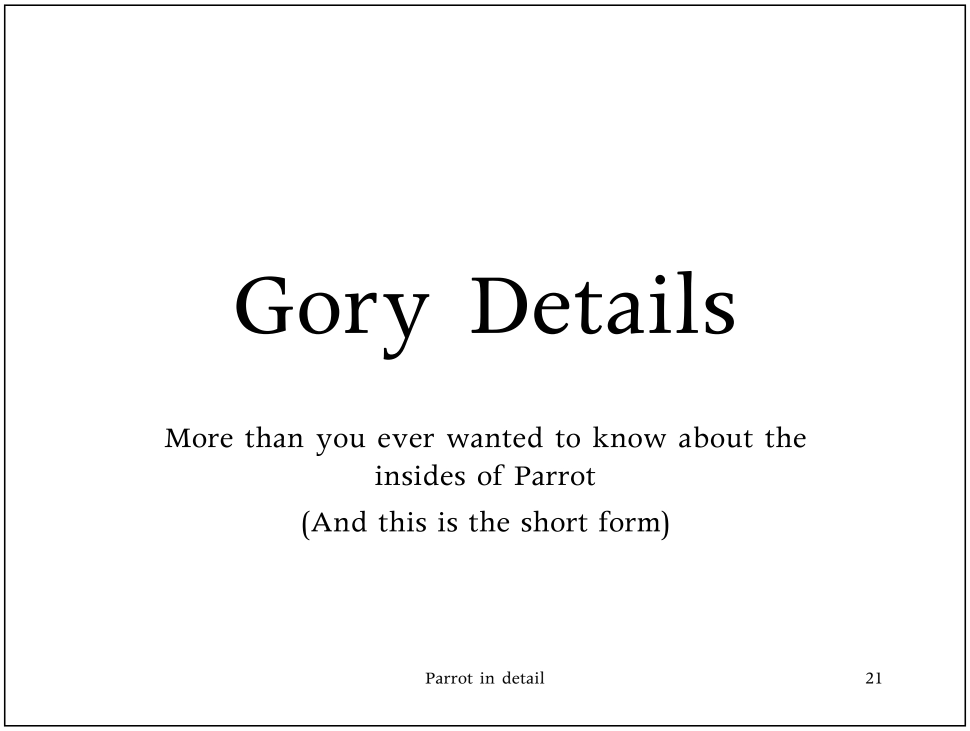


Interpreter

- As final destination, may do other things
  - Save to disk
  - Transform to an alternate form (JVM, .NET)
  - JIT
  - Mock in iambic pentameter

Parrot in detail 20

Needless to say, “Damian!” is the first thing that someone pipes up when I hit the final point. :)



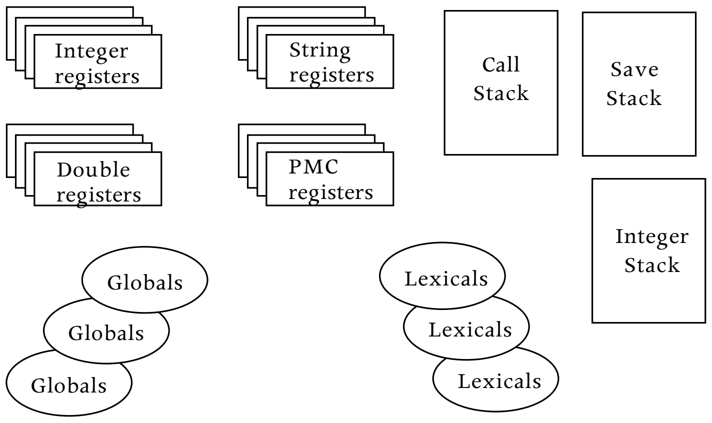
Gory Details

More than you ever wanted to know about the insides of Parrot

(And this is the short form)

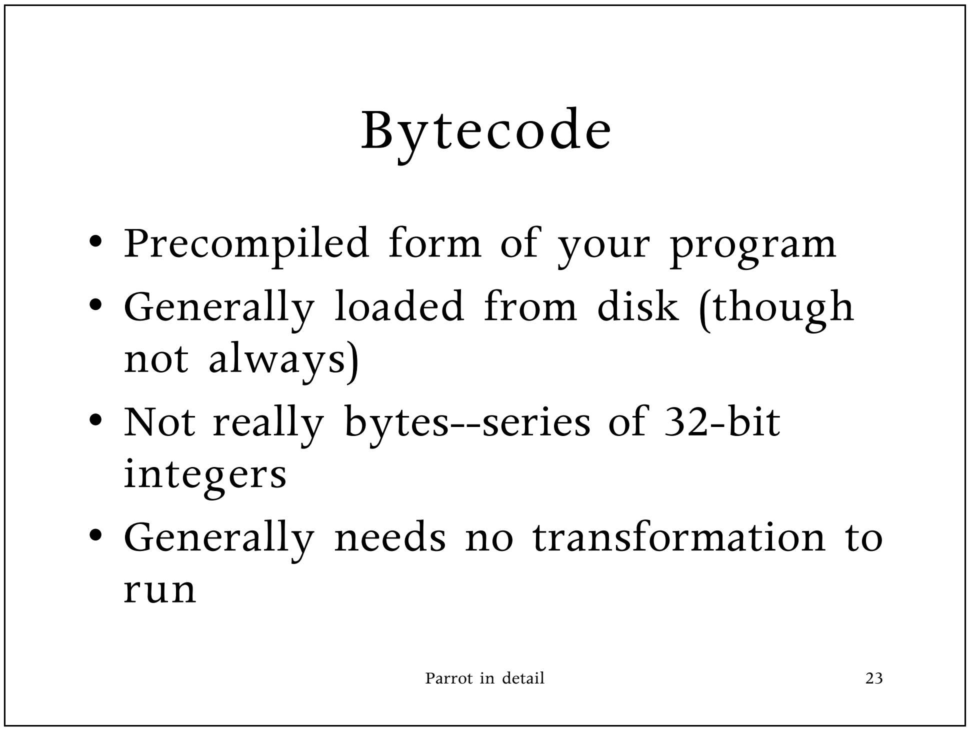
And now on to the details of the pieces

The internal architecture



Parrot in detail 22

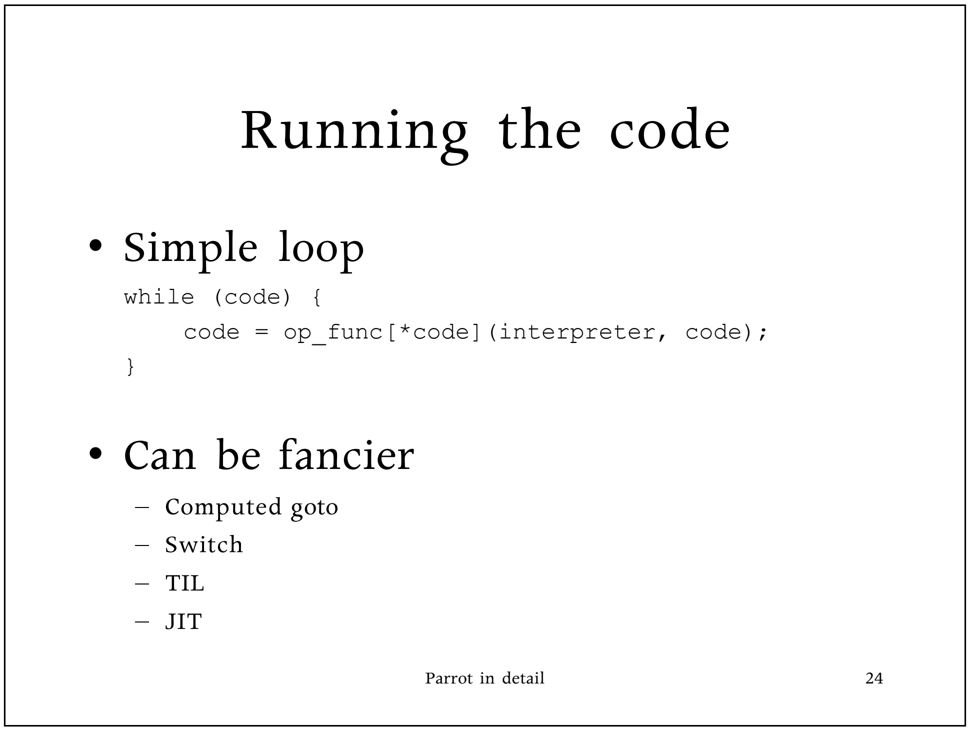
It seems so innocuous… If this were a CPU diagram there'd be active components here, like ALUs and FPUs. We don't really have that.



Bytecode

- Precompiled form of your program
- Generally loaded from disk (though not always)
- Not really bytes--series of 32-bit integers
- Generally needs no transformation to run

An interesting side note--because we have such a huge range of opcode numbers, and because we have the capability to load in opcode functions as we choose, and because the bytecode loader may do a transform and is pluggable, we can run JVM and .NET code directly.



Running the code

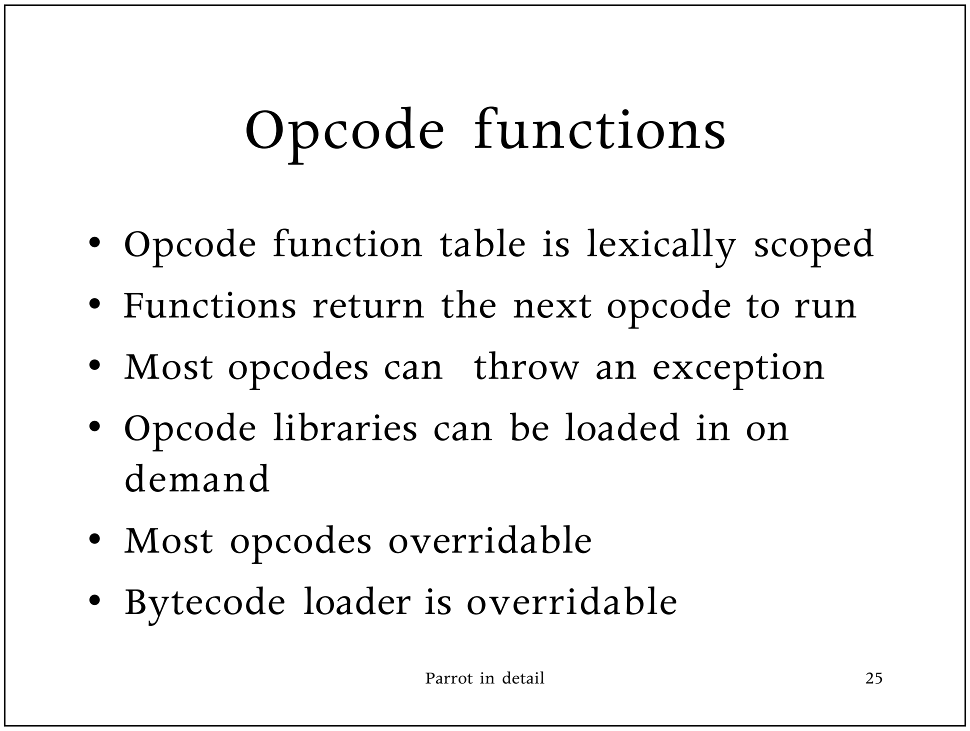
- Simple loop

```
while (code) {
    code = op_func[*code](interpreter, code);
}
```

- Can be fancier
  - Computed goto
  - Switch
  - TIL
  - JIT

Parrot in detail 24

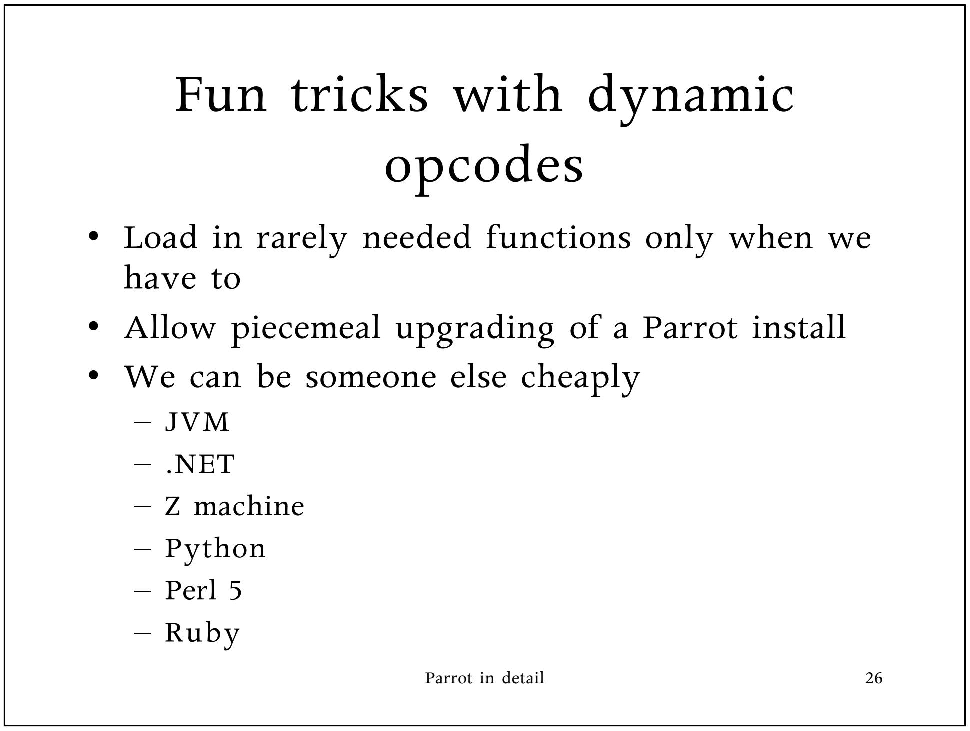
Error checking's not here, of course. It should be, but it's removed for simplicity and slide space.



Opcode functions

- Opcode function table is lexically scoped
- Functions return the next opcode to run
- Most opcodes can throw an exception
- Opcode libraries can be loaded in on demand
- Most opcodes overridable
- Bytecode loader is overridable

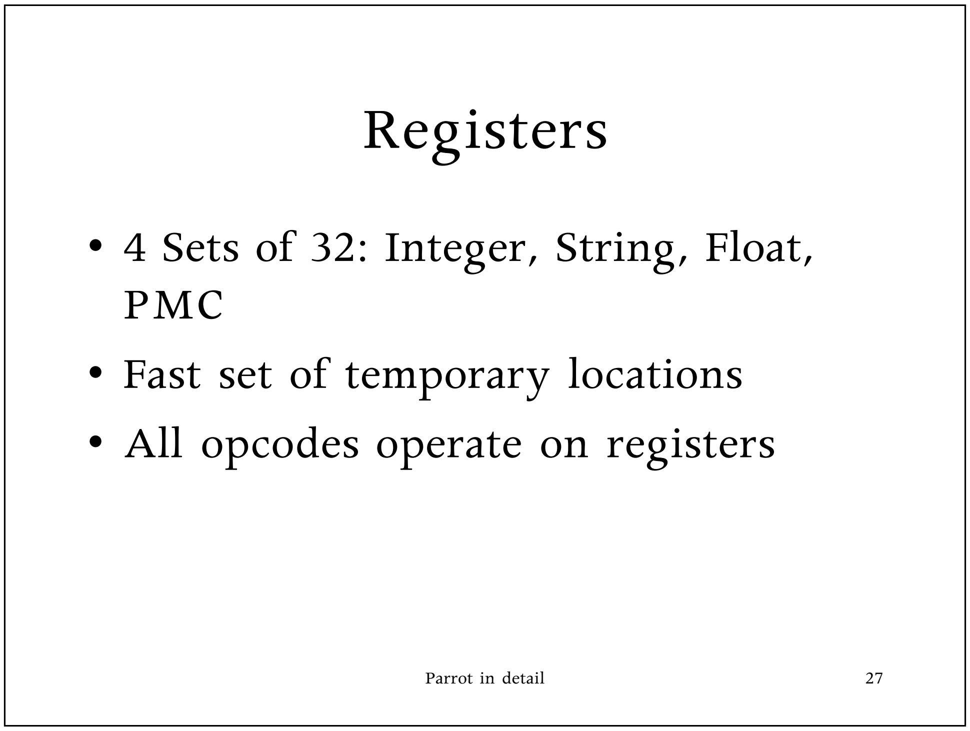
The core’s not 100% flexible--the core ops are fixed. Most everything else can be overridden, though.


Fun tricks with dynamic opcodes

- Load in rarely needed functions only when we have to
- Allow piecemeal upgrading of a Parrot install
- We can be someone else cheaply
  - JVM
  - .NET
  - Z machine
  - Python
  - Perl 5
  - Ruby

Parrot in detail 26

Parrot *will* be able to run z machine games by the time we release, dammit!. I have the data files for Lurking Horror, and I’m not afraid to use them!

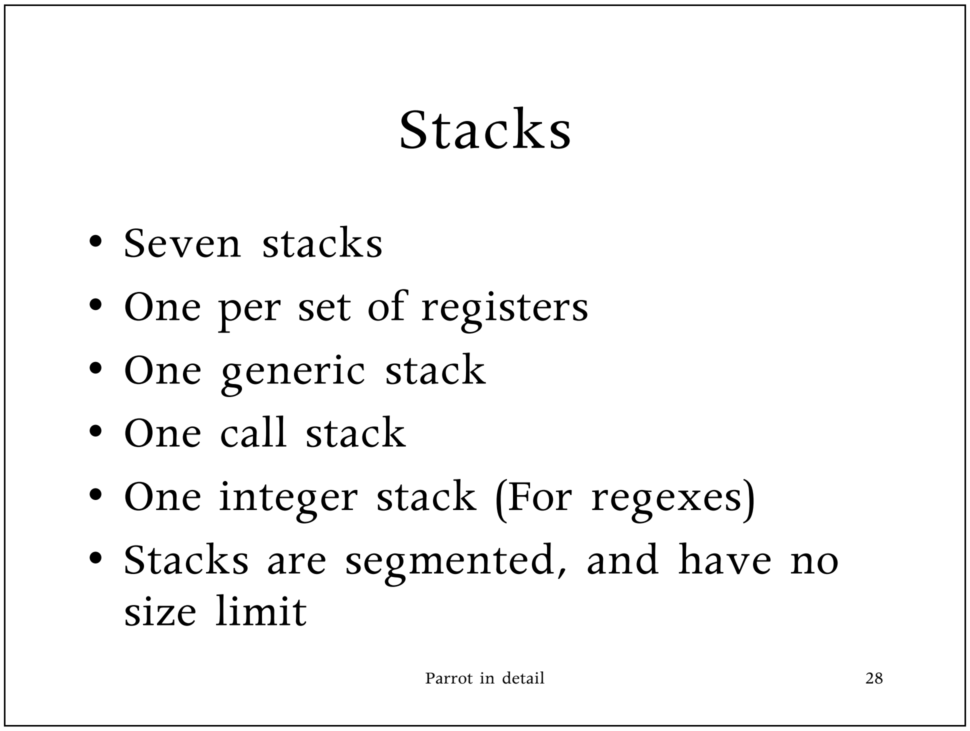


Registers

- 4 Sets of 32: Integer, String, Float, PMC
- Fast set of temporary locations
- All opcodes operate on registers

Parrot in detail 27

Note that the int, string, and float registers are mostly for internal use.

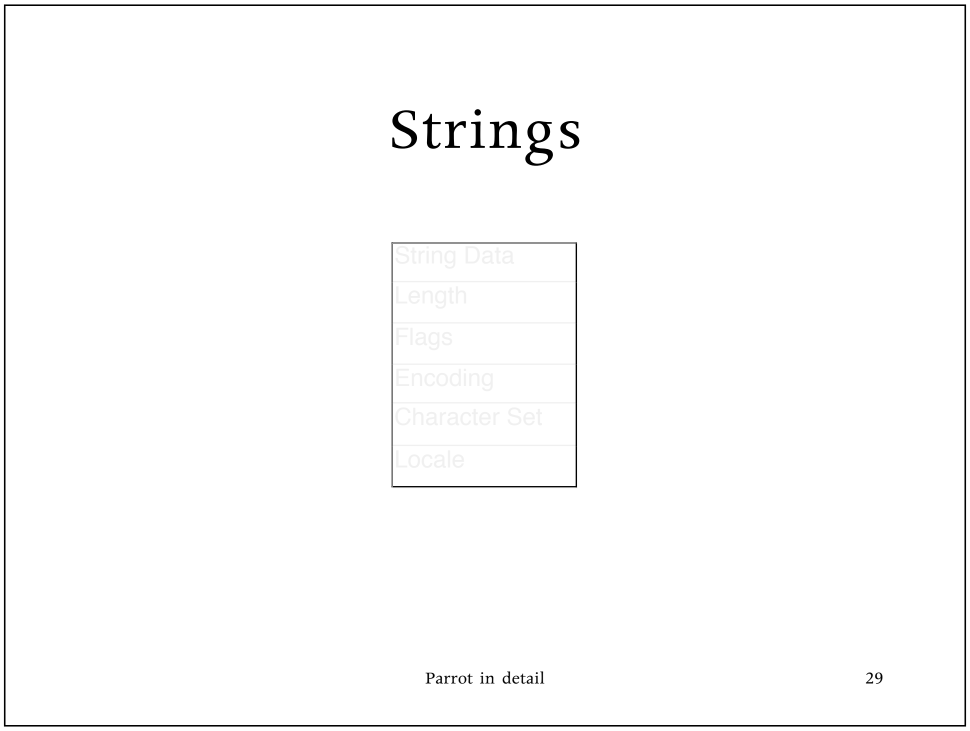


Stacks

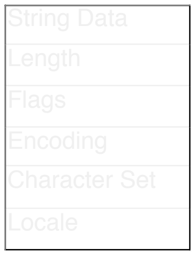
- Seven stacks
- One per set of registers
- One generic stack
- One call stack
- One integer stack (For regexes)
- Stacks are segmented, and have no size limit

Parrot in detail 28

Yes, we have stacks, gobs and gobs of stacks.



Strings

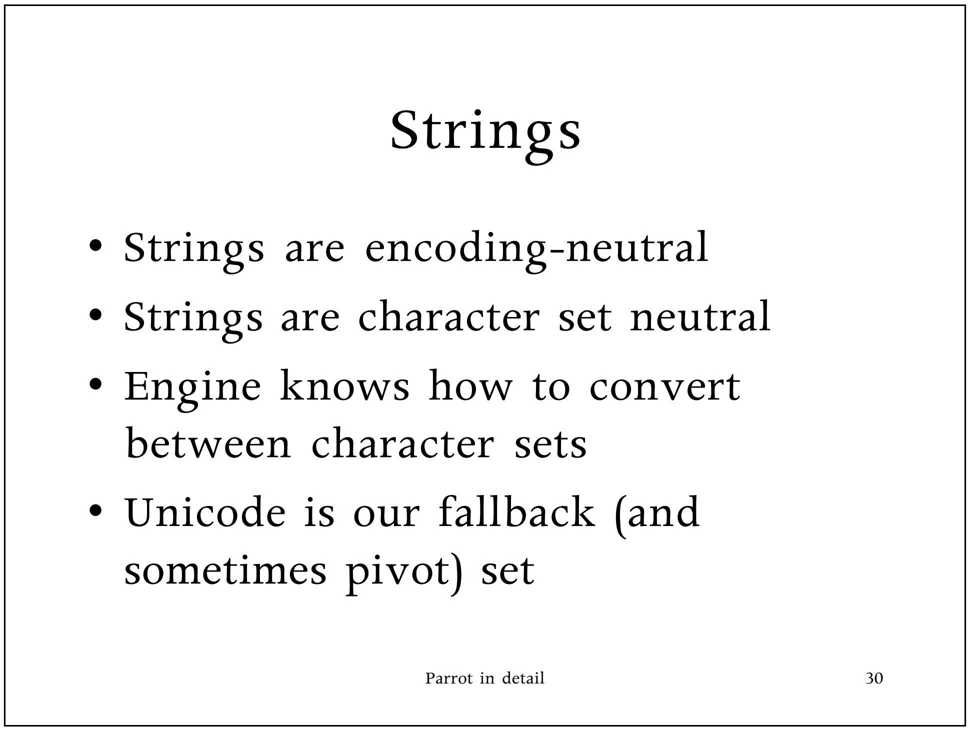


Parrot in detail 29

There's actually a bit more to strings, since we do COW for strings and substrings, but this is it.

Locale's sort of "language string came from" when we know it. We need this for proper string sorting--different languages sort differently. "ll" or an accented vowel sort differently depending on the language/locale the string came from.

I usually make it a point to point out that software should bow to people, not the other way around. I find it rather repulsive that people will change the rules of their language to match the limitations of a computer program.

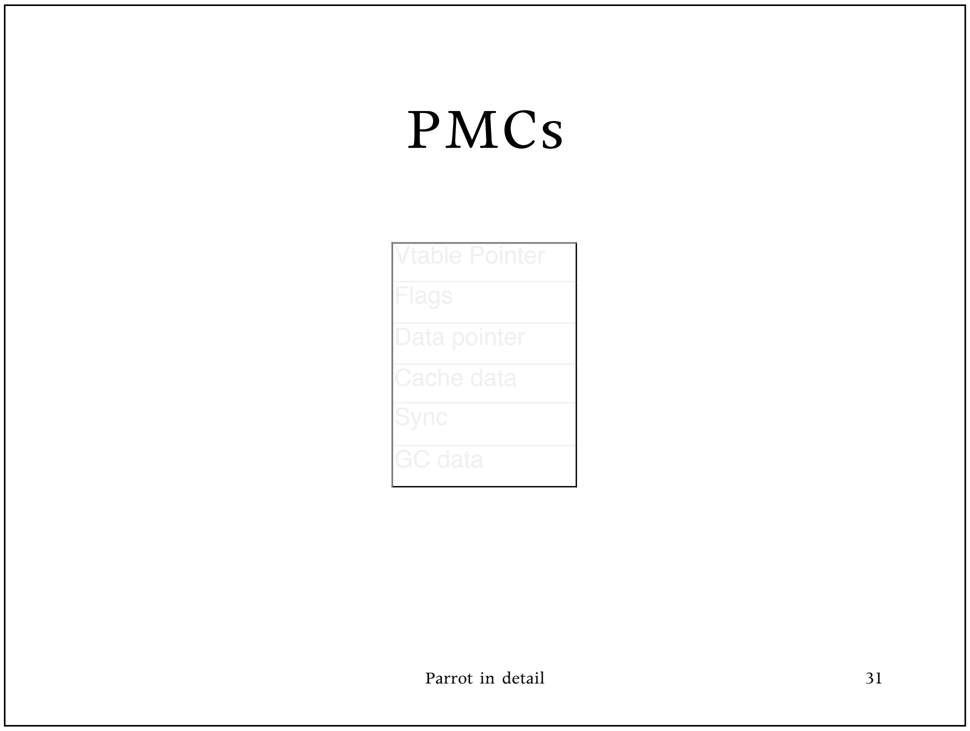


Strings

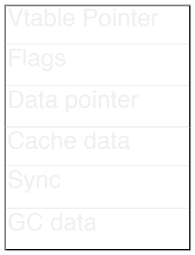
- Strings are encoding-neutral
- Strings are character set neutral
- Engine knows how to convert between character sets
- Unicode is our fallback (and sometimes pivot) set

Parrot in detail 30

While Perl 6 may be unicode all the way, Parrot is *not*. It doesn't give a rip, really, and won't. There's no real reason why it should, ultimately. We've a slight bias towards unicode as a fallback, but unicode conversion's lossy.

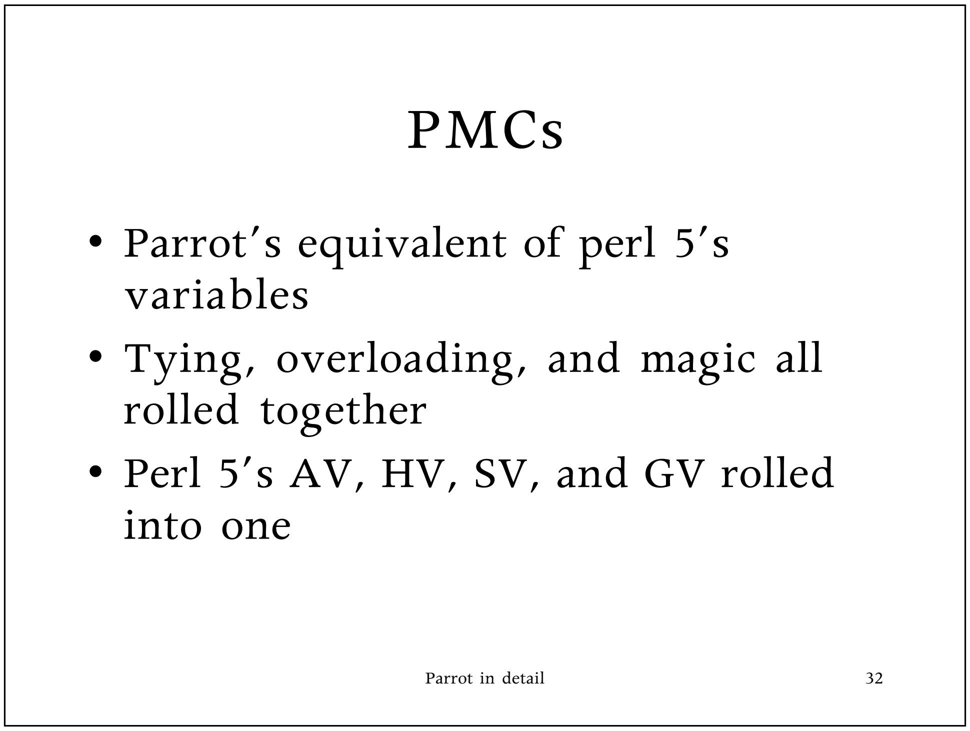


PMCs



Parrot in detail 31

Kinda small, isn't it?



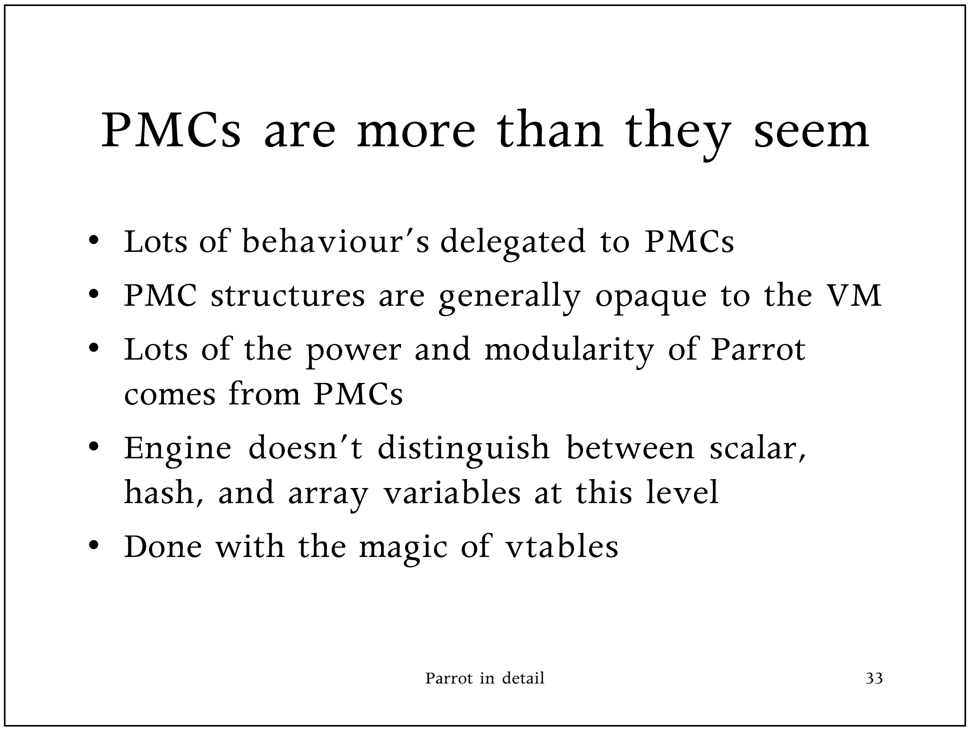
PMCs

- Parrot's equivalent of perl 5's variables
- Tying, overloading, and magic all rolled together
- Perl 5's AV, HV, SV, and GV rolled into one

Parrot in detail 32

And yet still faster than perl 5. Go figure. :)

Because of this architecture you pay for fancy features when accessing the data that uses it, not the operators that might be overridden. Perl 5 checks every variable for magic or tying on every access, and checks every variable to see if it overloads an operator every time it performs an operation. Yech. Much wasted time, there.

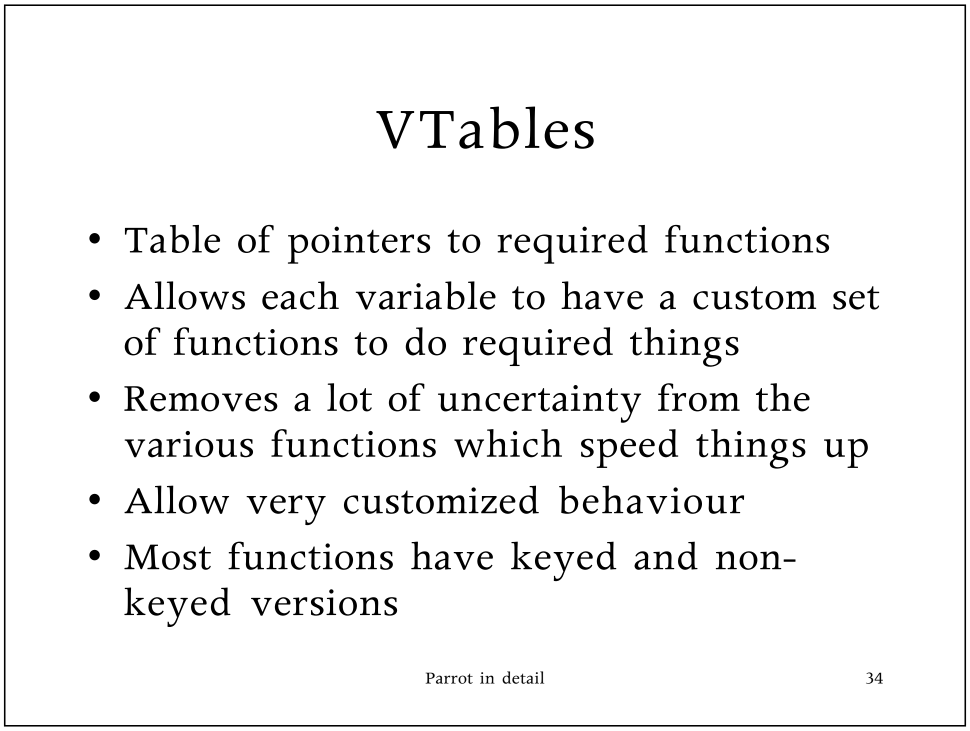


PMCs are more than they seem

- Lots of behaviour's delegated to PMCs
- PMC structures are generally opaque to the VM
- Lots of the power and modularity of Parrot comes from PMCs
- Engine doesn't distinguish between scalar, hash, and array variables at this level
- Done with the magic of vtables

Parrot in detail 33

PMCs are both wonderfully complex and nicely simplifying. They let us punt on most variable actions and let the variable decide what to do, which is what really should be done. We can have more special-case straight-line code that way.



VTables

- Table of pointers to required functions
- Allows each variable to have a custom set of functions to do required things
- Removes a lot of uncertainty from the various functions which speed things up
- Allow very customized behaviour
- Most functions have keyed and non-keyed versions

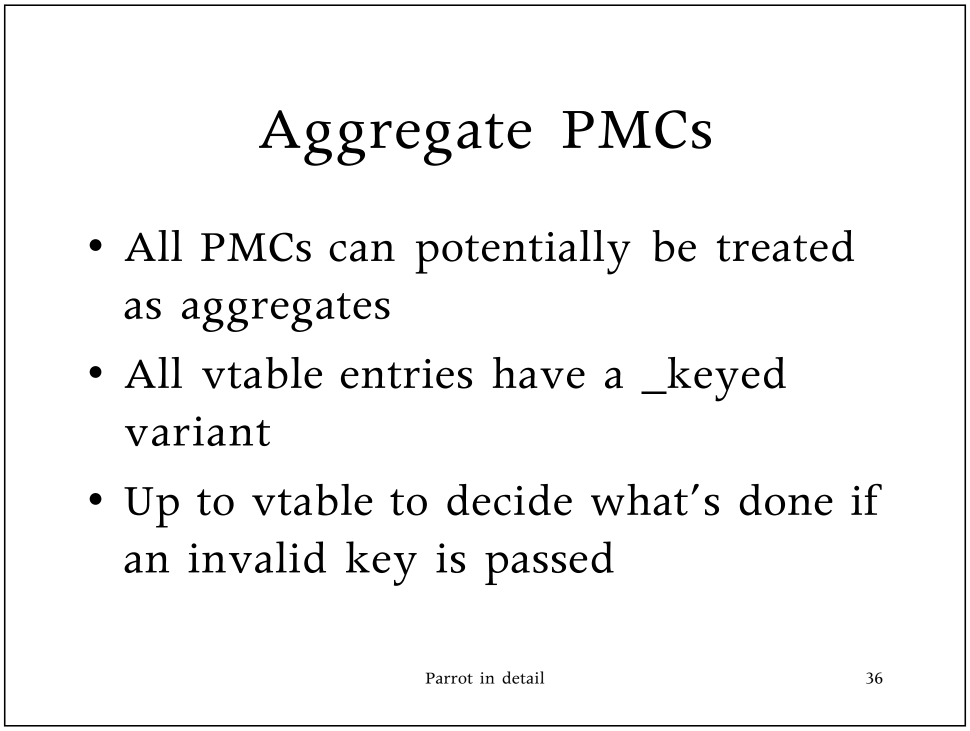
Parrot in detail 34

The third point's important. Branches are *expensive* on most processors these days, and each test's a branch. Removing the tests speeds things up a lot.

Vtables

- Some example functions

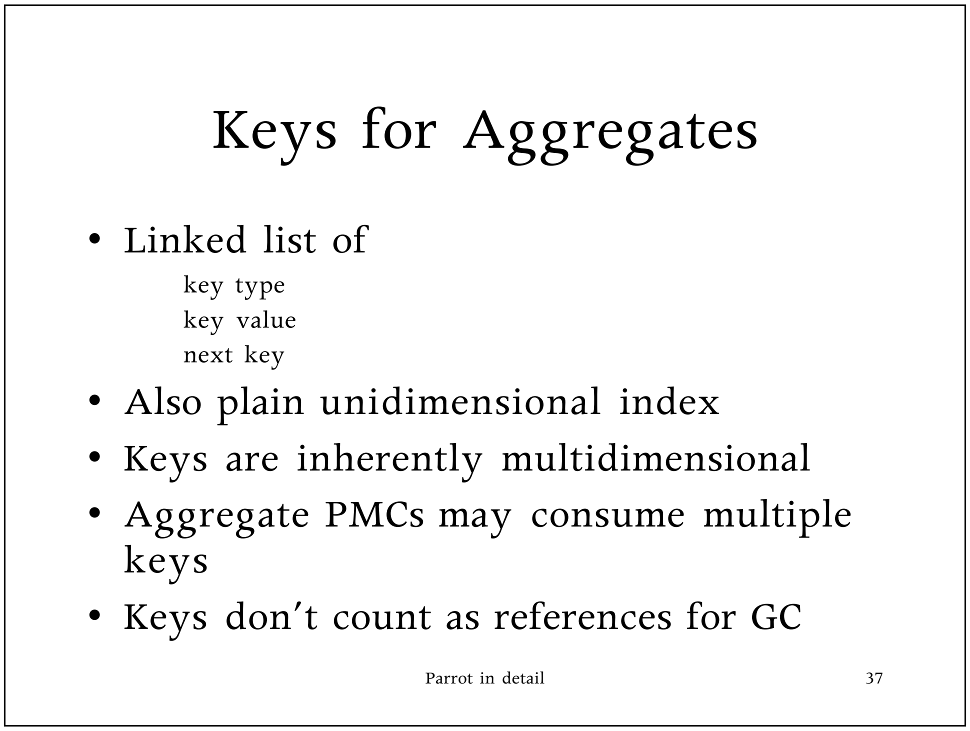
  name
  type
  clone
  get_(integer|float|string|value)
  set_(integer|float|string|value)
  add, subtract, multiply, divide
  call method
  GC special methods



Aggregate PMCs

- All PMCs can potentially be treated as aggregates
- All vtable entries have a _keyed variant
- Up to vtable to decide what's done if an invalid key is passed

The engine cares remarkably little about whether a PMC is an aggregate or not.

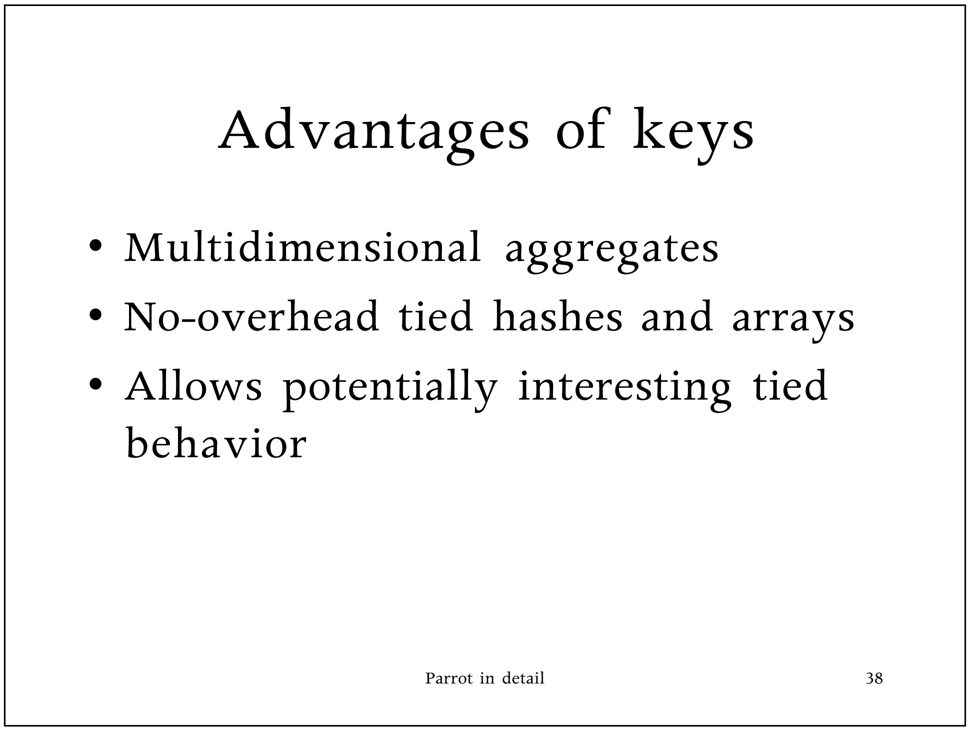


Keys for Aggregates

- Linked list of
  - key type
  - key value
  - next key
- Also plain unidimensional index
- Keys are inherently multidimensional
- Aggregate PMCs may consume multiple keys
- Keys don't count as references for GC

Parrot in detail 37

The structure may change, I'm not sure. Doing it this way means we can have true multidimensional arrays/hashes/whatever.

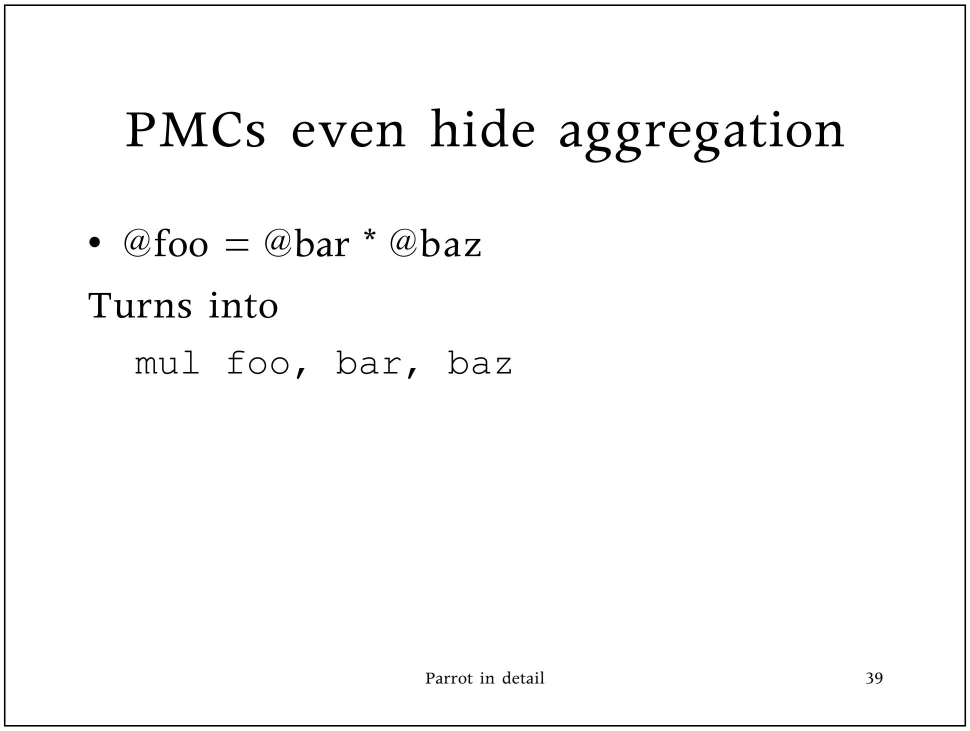


Advantages of keys

- Multidimensional aggregates
- No-overhead tied hashes and arrays
- Allows potentially interesting tied behavior

Parrot in detail 38

The potentially interesting tied behaviour is being able to tell whether you're calling @pid[$$]{time} or @pid[$$]{time}{user}, in case you care.



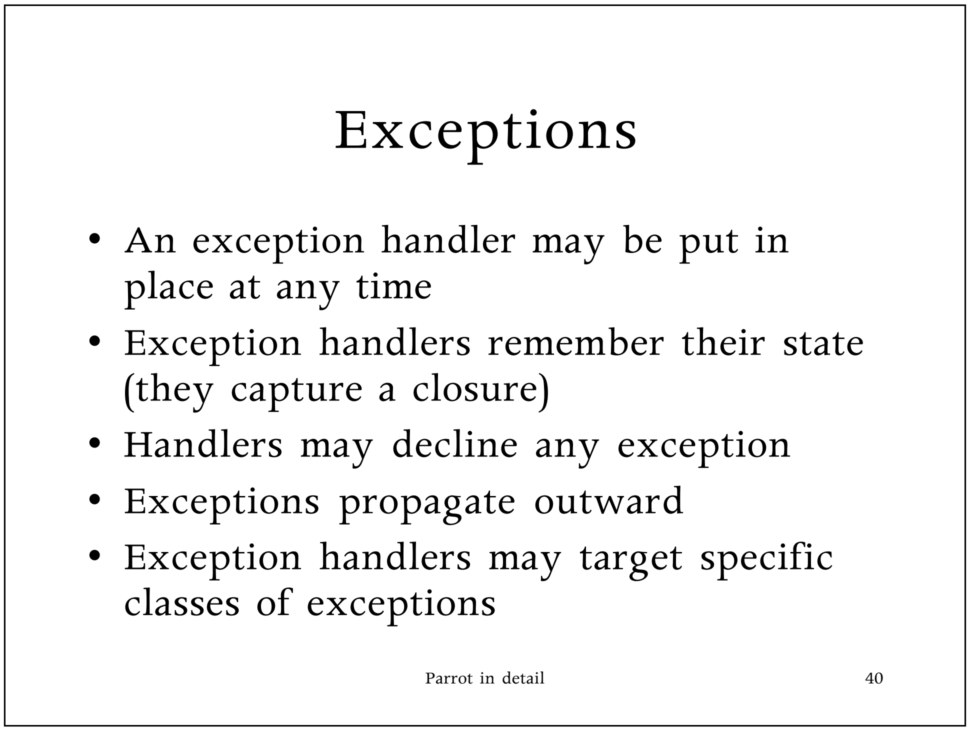
PMCs even hide aggregation

- @foo = @bar * @baz

Turns into

```
mul foo, bar, baz
```

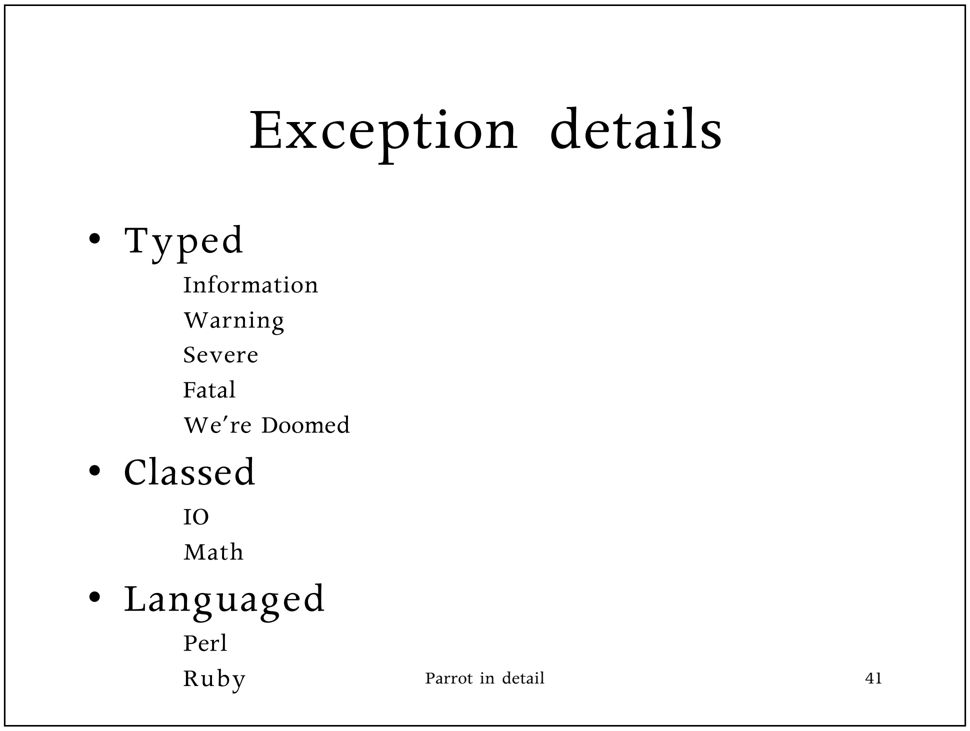
Means that aggregate PMCs can do something interesting when treated as a single thing. Matrix math, say.


Exceptions

- An exception handler may be put in place at any time
- Exception handlers remember their state (they capture a closure)
- Handlers may decline any exception
- Exceptions propagate outward
- Exception handlers may target specific classes of exceptions

Parrot in detail 40

But they don’t resume, which is probably for the best. Allowing resumable exceptions is a major pain.

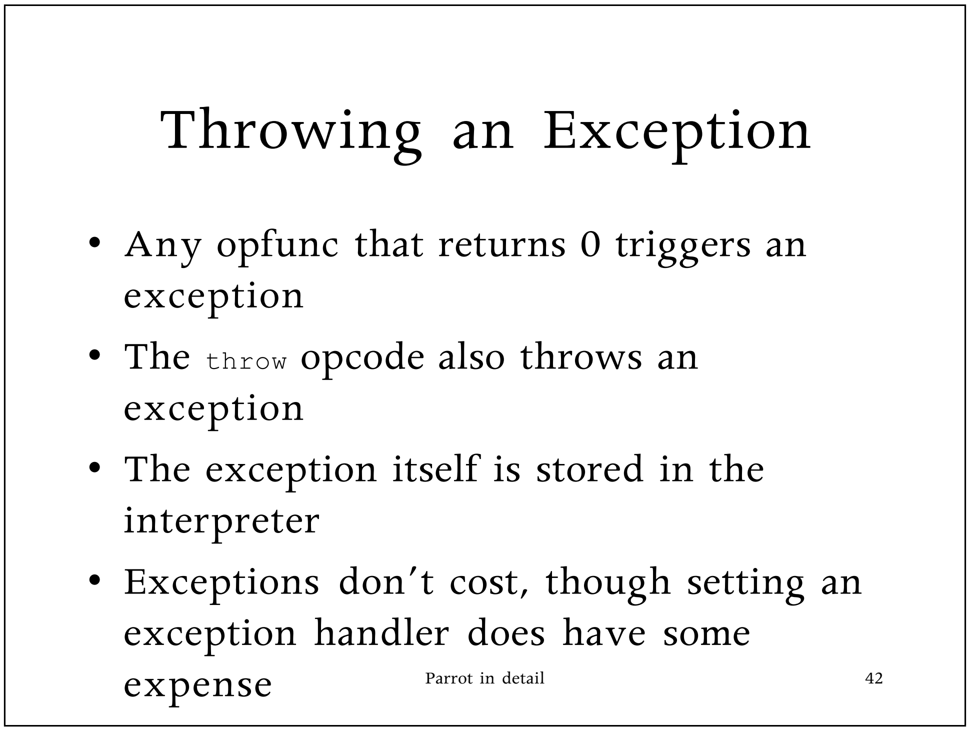


Exception details

- Typed
  - Information
  - Warning
  - Severe
  - Fatal
  - We're Doomed
- Classed
  - IO
  - Math
- Languaged
  - Perl
  - Ruby

Parrot in detail 41

Yes, exceptions will carry the language that was in effect when they were thrown.

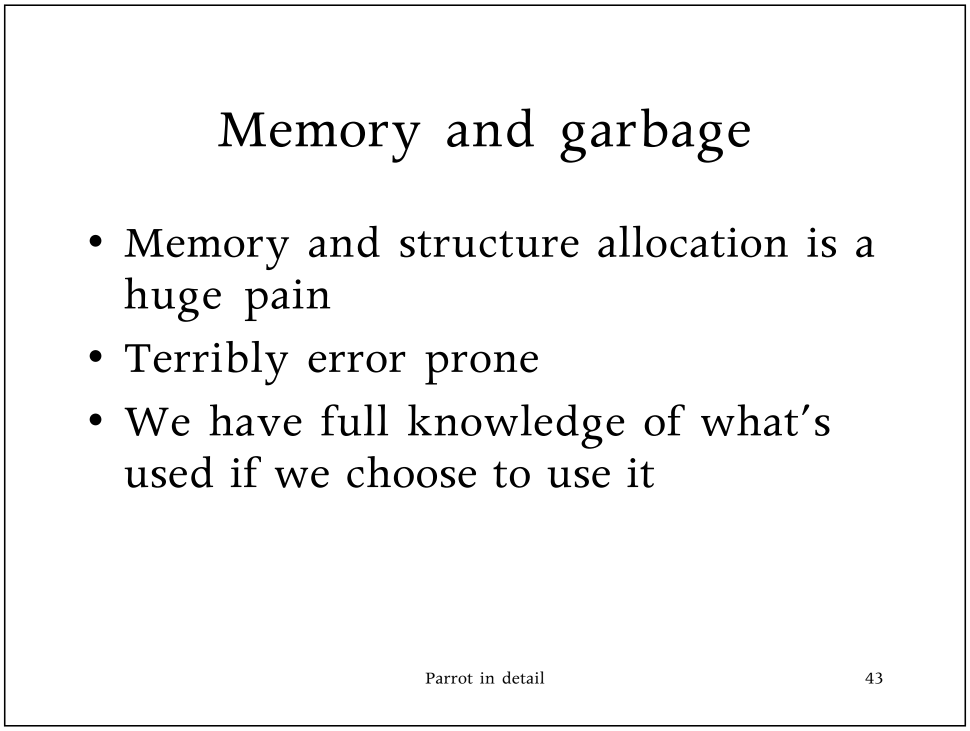


Throwing an Exception

- Any opfunc that returns 0 triggers an exception
- The `throw` opcode also throws an exception
- The exception itself is stored in the interpreter
- Exceptions don't cost, though setting an exception handler does have some expense

Parrot in detail 42

We generally use longjmp for exception throwing. We do a setjmp outside the runloop (returning 0 exits the runloop, hence that is an exception too) and just let it sit until we need it.



Memory and garbage

- Memory and structure allocation is a huge pain
- Terribly error prone
- We have full knowledge of what's used if we choose to use it

Parrot in detail 43

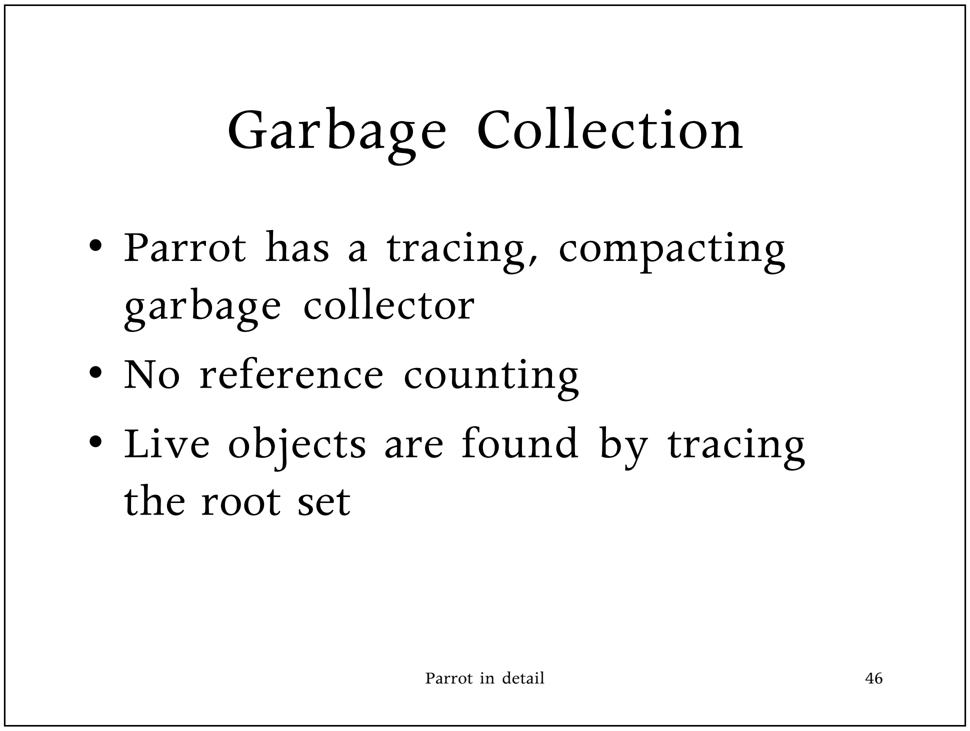
Memory allocation's a major waste of time. In more than one sense.

Arena Allocation of core structures

- All PMCs and Strings are allocated from arenas
- Makes allocation faster and more memory efficient
- Allows us to trace all the core structures as we need for GC and DOD

Pool allocation of memory

- All 'random' chunks of memory are allocated from memory pools
- Allocation is extremely fast, typically five or six machine instructions
- Free memory is handled by the garbage collector

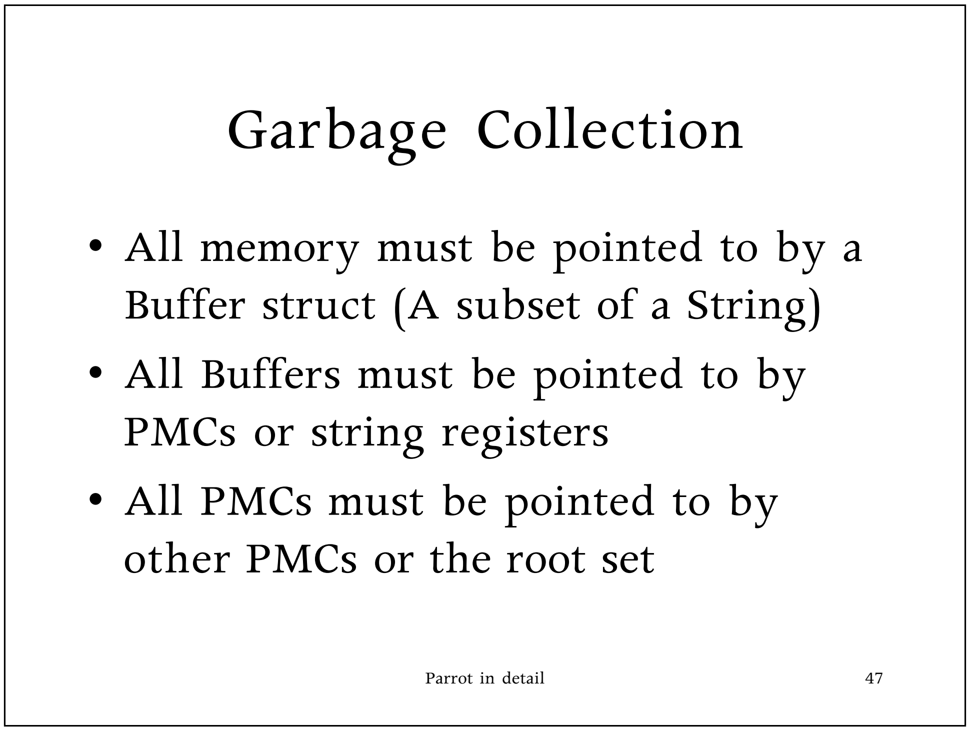


Garbage Collection

- Parrot has a tracing, compacting garbage collector
- No reference counting
- Live objects are found by tracing the root set

Parrot in detail 46

Circular references are OK now--we find those. And no, there’s no guarantee of destruct order for objects with an active destructor. The destruction phase can be overridden if you want to impose some sort of order, but that’s tricky.

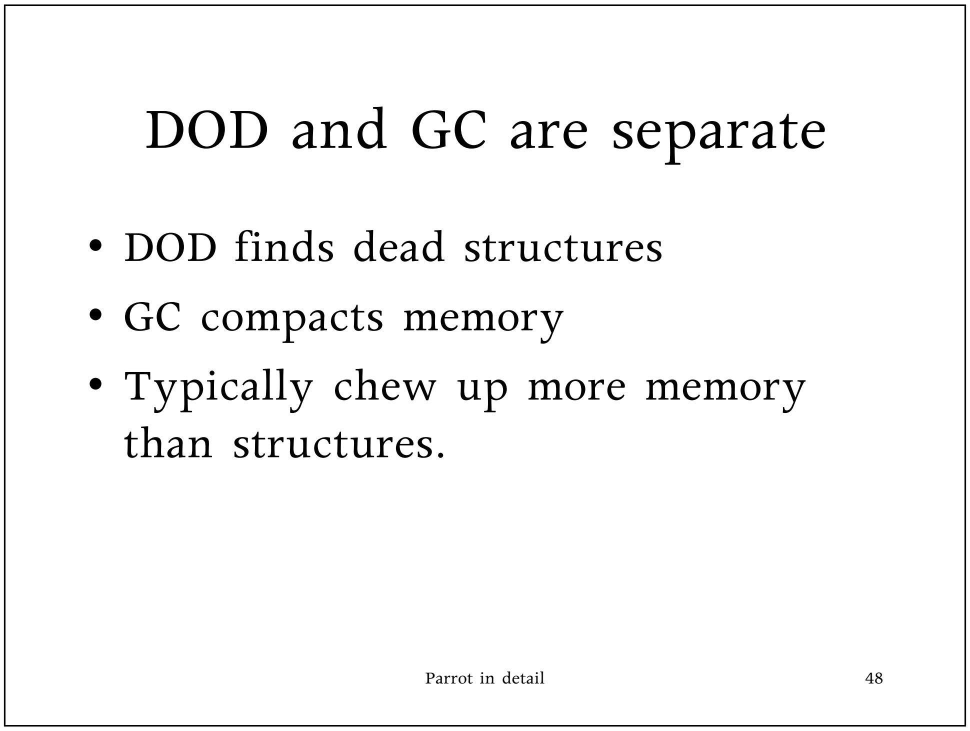


Garbage Collection

- All memory must be pointed to by a Buffer struct (A subset of a String)
- All Buffers must be pointed to by PMCs or string registers
- All PMCs must be pointed to by other PMCs or the root set

Parrot in detail 47

A bit restrictive, but it makes things easier. And easier tends to be faster. Besides, it's less work to lift what turns out to be an onerous restriction than to put a restriction in place after the fact.

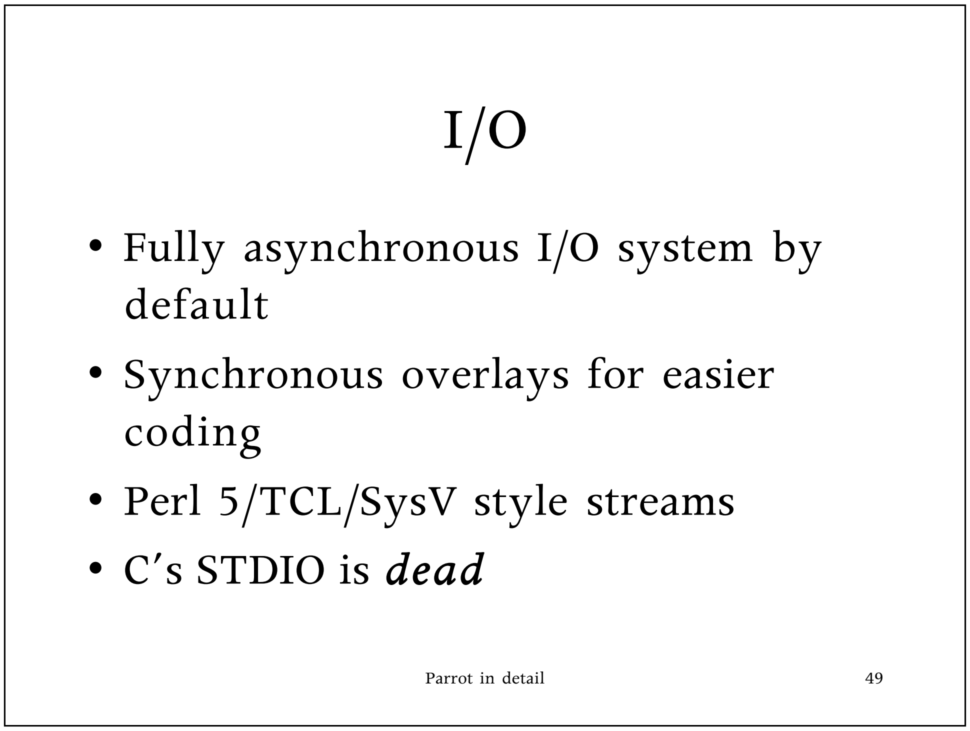


DOD and GC are separate

- DOD finds dead structures
- GC compacts memory
- Typically chew up more memory than structures.

Parrot in detail 48

Since we’ve mutable strings, the third point is often very true for perl.



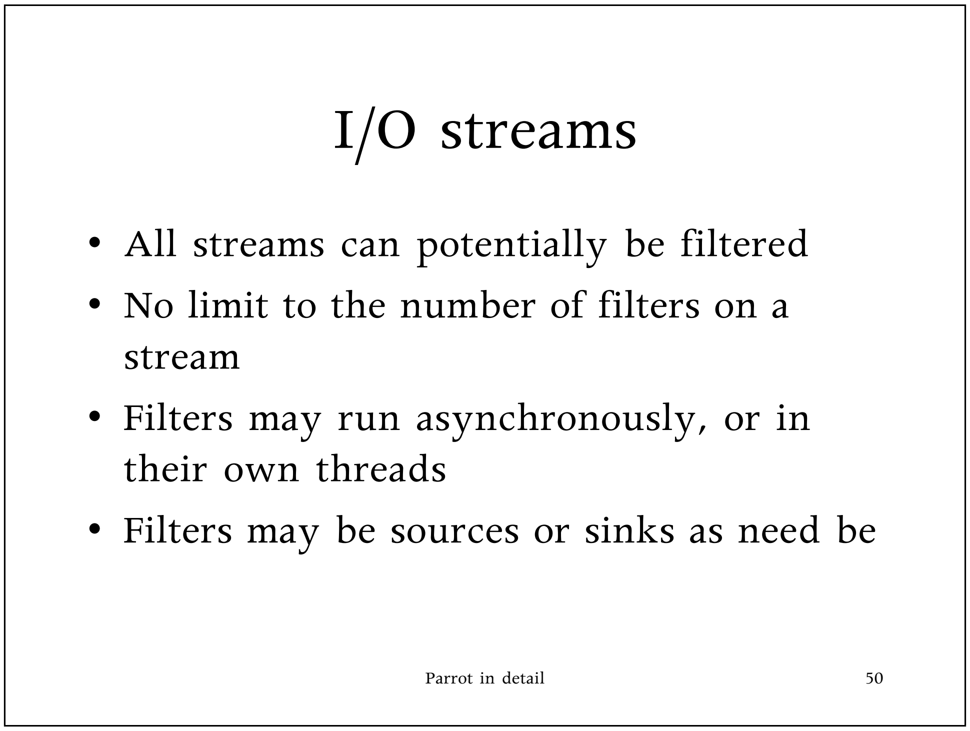
I/O

- Fully asynchronous I/O system by default
- Synchronous overlays for easier coding
- Perl 5/TCL/SysV style streams
- C's STDIO is ***dead***

Parrot in detail 49

Layering synchrony on an asynchronous system's easy. Layering asynchrony on a synchronous system's a major pain in the neck.

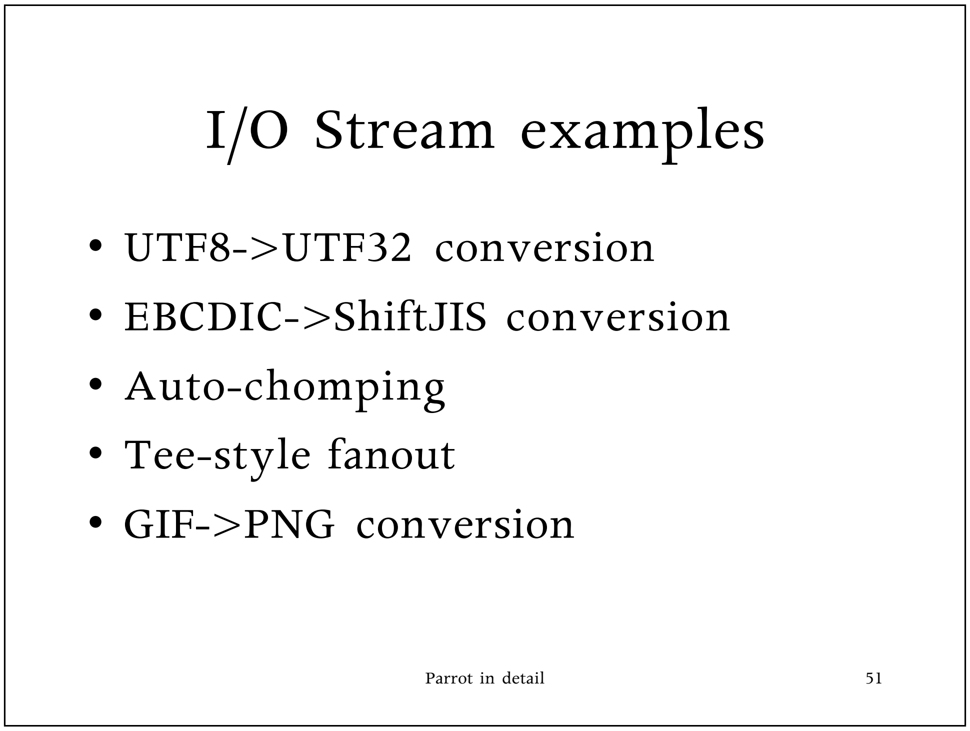
The last point usually gets applause from folks who've had it inflicted on them for any length of time.



I/O streams

- All streams can potentially be filtered
- No limit to the number of filters on a stream
- Filters may run asynchronously, or in their own threads
- Filters may be sources or sinks as need be

Yes, very Tcl-ish.

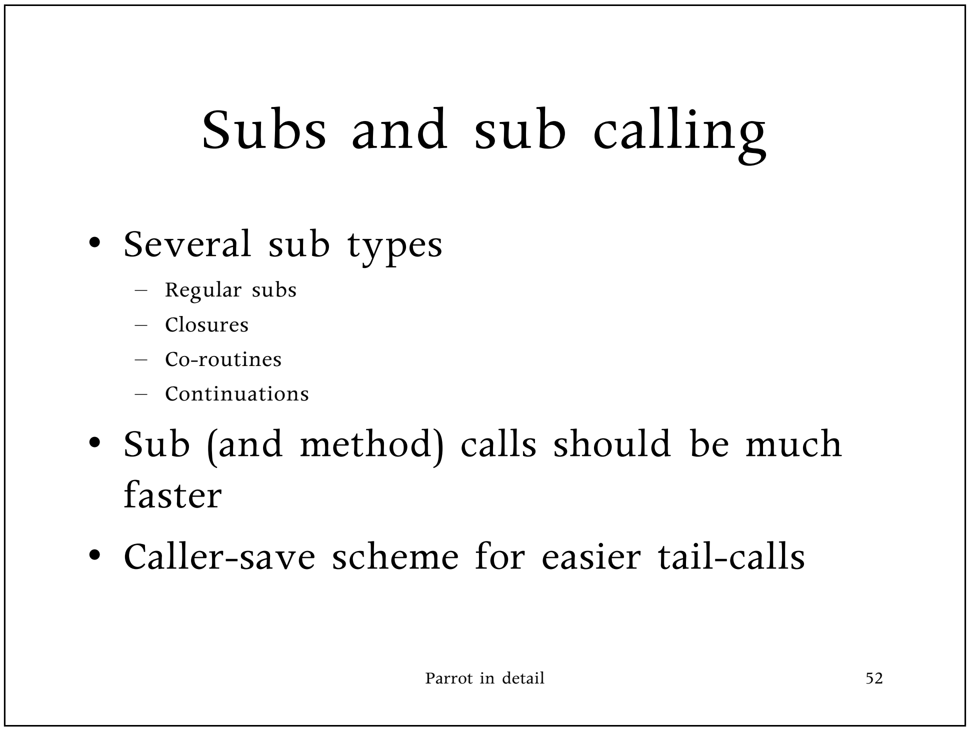


I/O Stream examples

- UTF8->UTF32 conversion
- EBCDIC->ShiftJIS conversion
- Auto-chomping
- Tee-style fanout
- GIF->PNG conversion

Parrot in detail 51

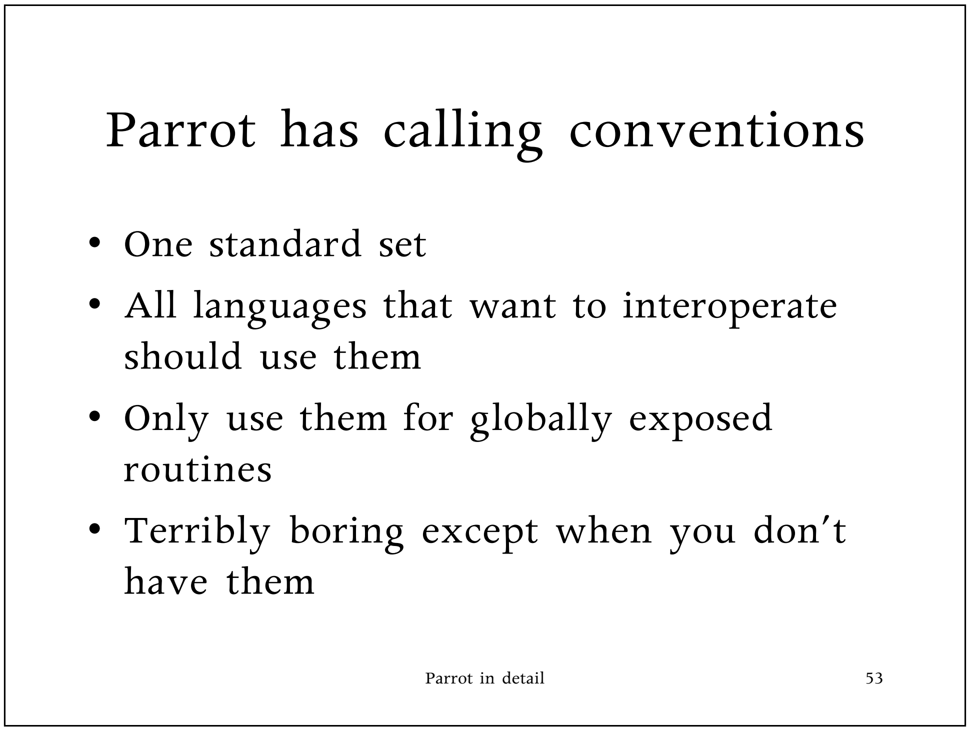
Or any number of things. Heck, you could treat an IO stream as just a black box data morpher if you so chose.



Subs and sub calling

- Several sub types
  - Regular subs
  - Closures
  - Co-routines
  - Continuations
- Sub (and method) calls should be much faster
- Caller-save scheme for easier tail-calls

Generally people have no idea what continuations are, or if they do it hurts to remember.

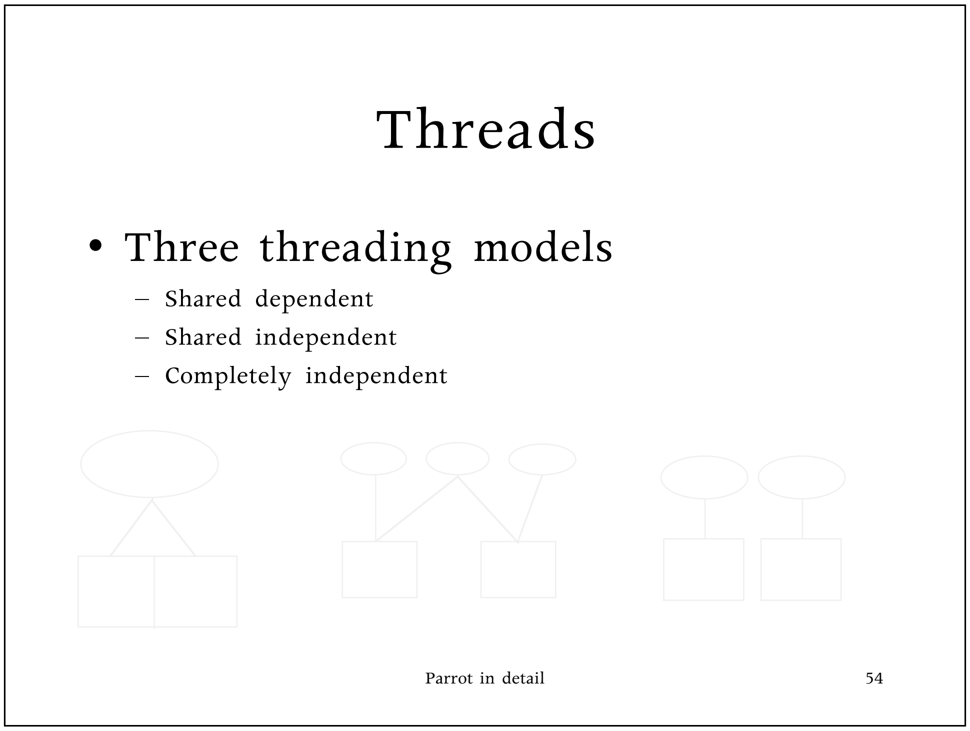


Parrot has calling conventions

- One standard set
- All languages that want to interoperate should use them
- Only use them for globally exposed routines
- Terribly boring except when you don't have them

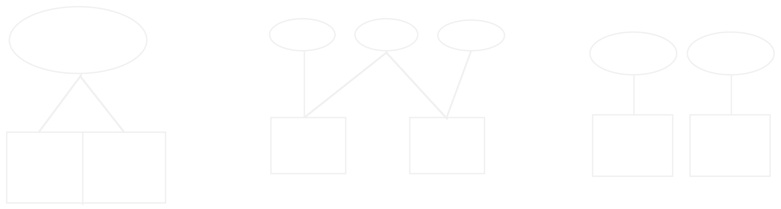
Parrot in detail 53

You don't realize how nice standard calling conventions are until you don't have them. There's nothing worse than having, say, three different Fortran compilers for the same system that generate completely incompatible object files because their calling conventions are all different.



Threads

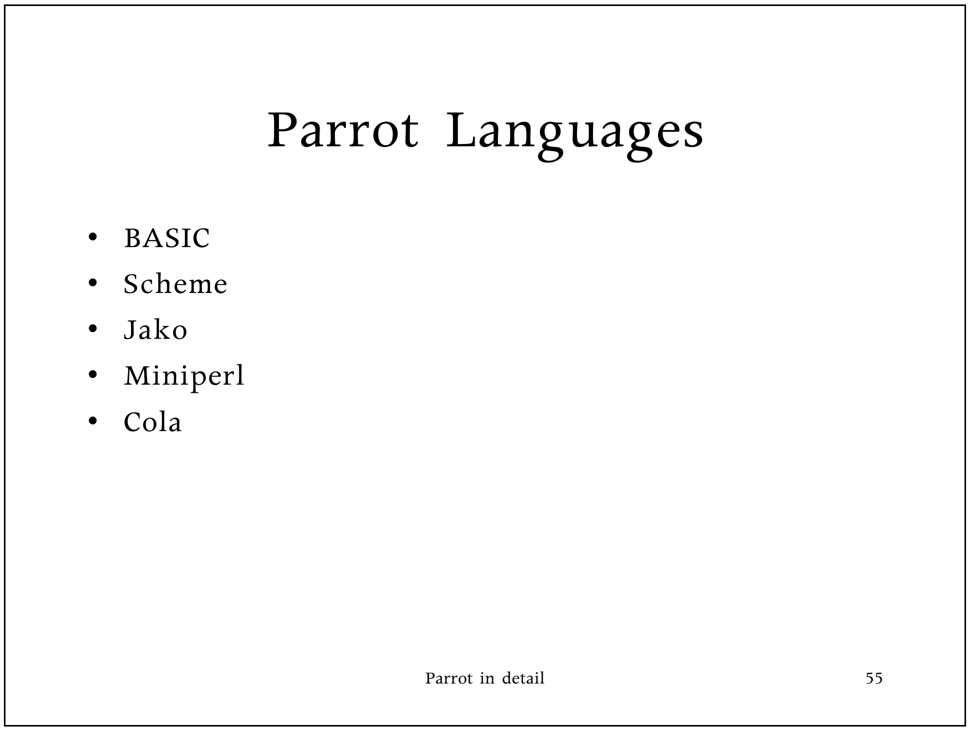
- Three threading models
  - Shared dependent
  - Shared independent
  - Completely independent



The ovals are variables and the squares are interpreters.

In the second and third diagrams, the interpreters are in separate threads and run independently. The data shared in the middle is automatically locked on access.

In the first example, the interpreters share everything, and only one runs at once to avoid stepping on toes. Generally good for coroutines and such.

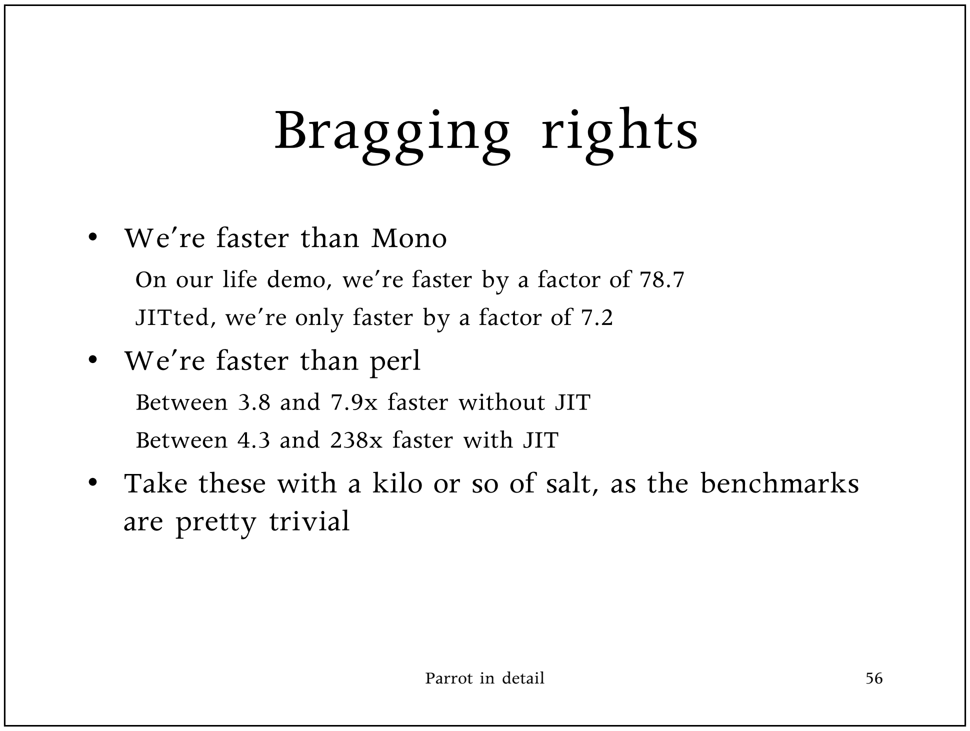


Parrot Languages

- BASIC
- Scheme
- Jako
- Miniperl
- Cola

Parrot in detail 55

And more coming, of course. The BASIC interpreter runs eliza and hunt the wumpus.



Bragging rights

- We're faster than Mono
  - On our life demo, we're faster by a factor of 78.7
  - JITted, we're only faster by a factor of 7.2
- We're faster than perl
  - Between 3.8 and 7.9x faster without JIT
  - Between 4.3 and 238x faster with JIT
- Take these with a kilo or so of salt, as the benchmarks are pretty trivial

Parrot in detail 56

The Mono folks have closed the gap a bit, and we've not benchmarked Microsoft's .NET because we don't have any hardware to do it on properly. (Simulations indicate Mono runs at about half the speed of MS' implementation, but I don't quote simulation numbers)