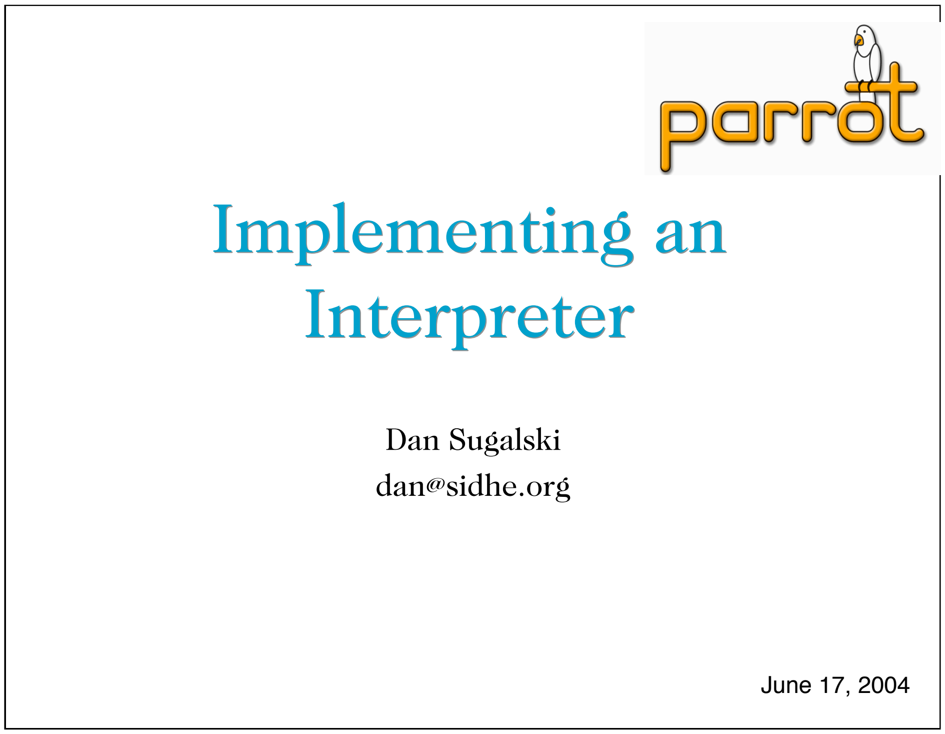
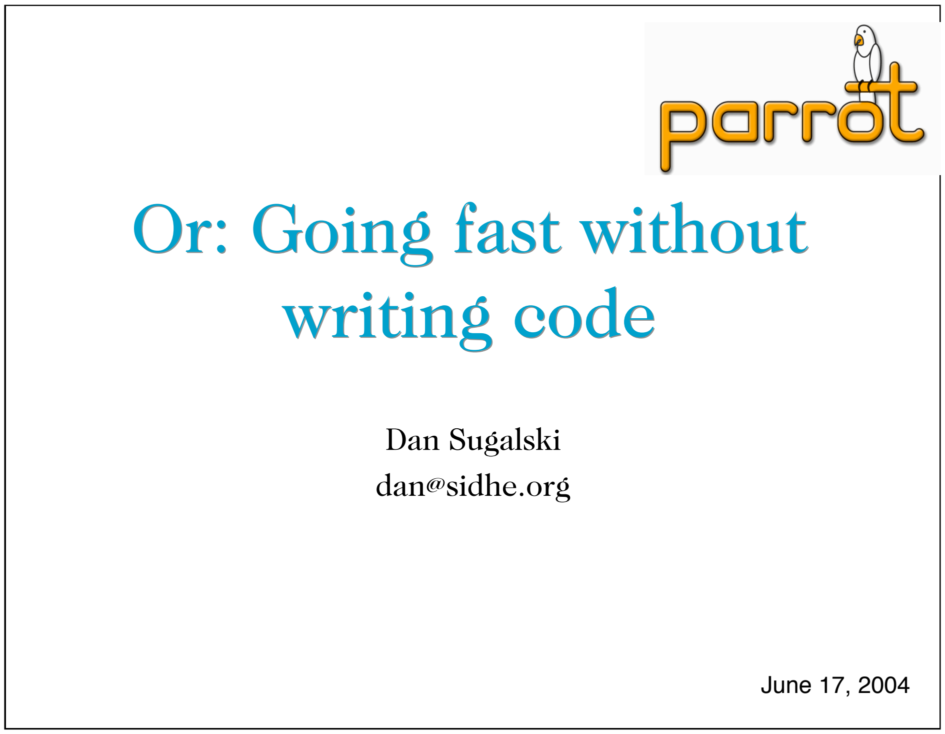


Implementing an Interpreter

Dan Sugalski
dan@sidhe.org

June 17, 2004

Well, sort of. This is more a “Here are some tricks and interesting things we did for parrot’s implementation”

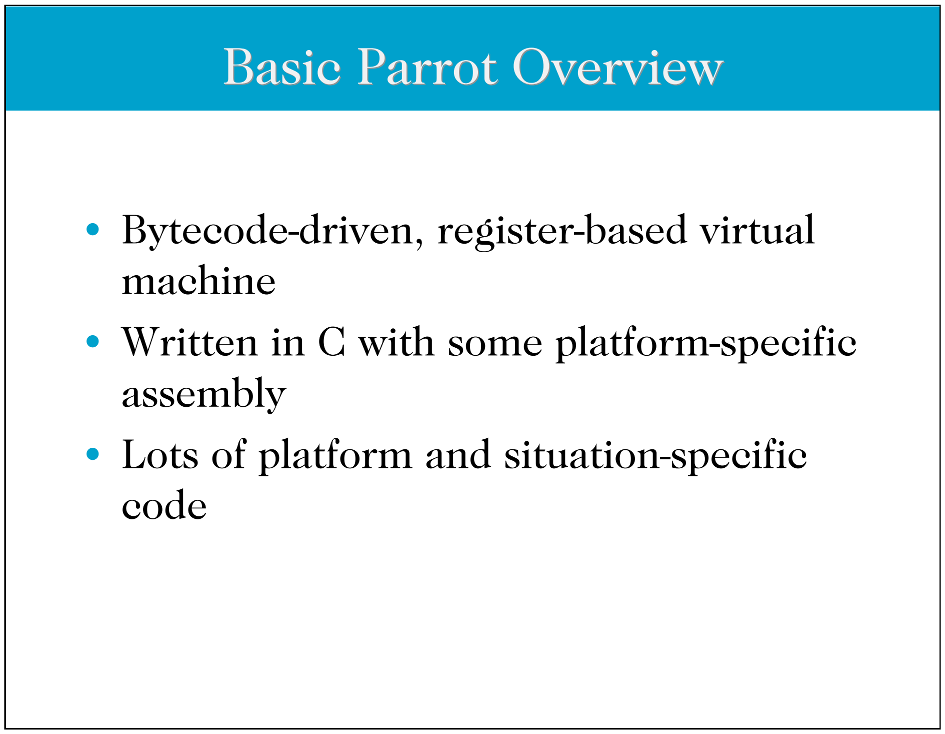


Or: Going fast without writing code

Dan Sugalski
dan@sidhe.org

June 17, 2004

Because the less code you write the fewer things you can screw up!



Basic Parrot Overview

- Bytecode-driven, register-based virtual machine
- Written in C with some platform-specific assembly
- Lots of platform and situation-specific code

Blah, blah, blah. Getting the buzzwords out of the way.

Parrot's written in C for two very big reasons. The first is because we can count on there being an adequate, if not actually good, C compiler everywhere. (GCC, while not great, is at least nearly ubiquitous) The second is that we can count on there being a good supply of C *programmers* nearly everywhere.

Parrot also grew out of Perl 5's world, where the source was all C, so the folks with experience doing this sort of thing were all fluent in C.

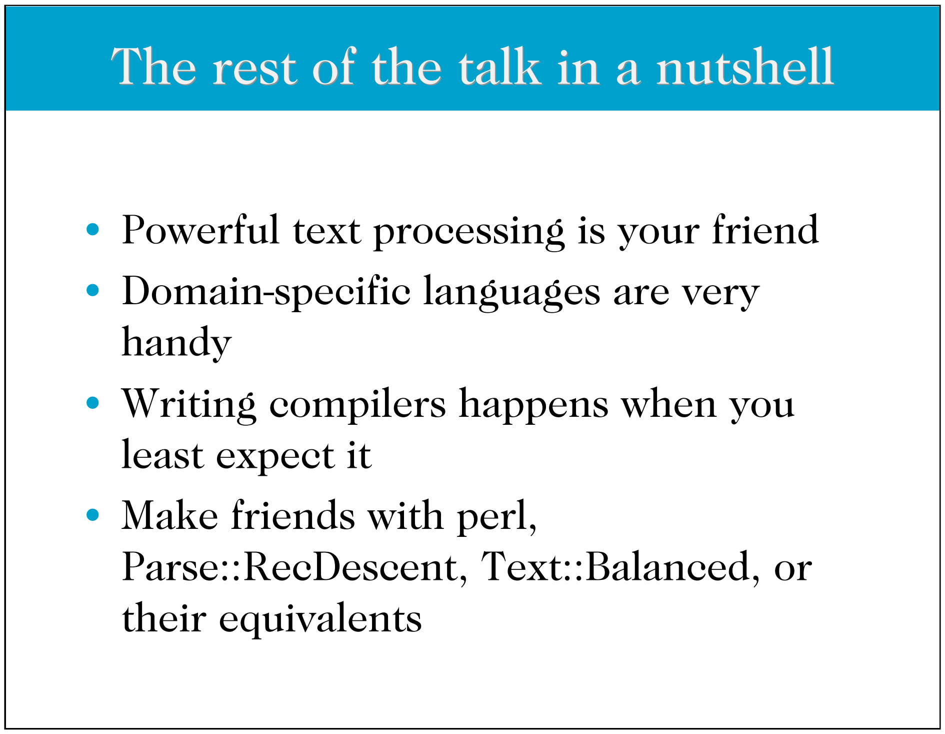
The platform specific stuff is just one of those facts of life when doing cross-platform code. You know, no matter what you might want, that at the very least you've got the Unix way(s) and Windows way to do most system tasks. Many of the different OSes also have private ways of doing things (asynchronous IO's a big one here) Not taking that into account at the beginning's a bad idea.

Since there's no real good spot to put this, here's where it goes: Why the heck is Parrot an interpreter anyway? Why not just go straight to JIT? Everybody JITs these days.

Simple question, not so simple answer. First, when Parrot was starting out, the only JITs around anyone knew about were for the JVM, and they were all done by scary-smart people who swore a lot about them. Everything else was interpreted. (If you throw VMs and interpreters into a big wad, which works for me)

Second, we were all coming from the perl (or possibly python or ruby) world, which is all interpreters. The folks involved (including me) aren't generally from deep in the CS world. About half the folks around could muster a BS, and that's about it, with most of us mostly self-taught. Optimizing compilers and JITs just aren't on the agenda, or at least they weren't.

Finally there's the portability issue. Interpreters are a lot more portable than JIT systems, and since perl runs pretty much everywhere we figured Parrot would too. (We didn't take into account the rapid and somewhat depressing attrition in the OS space)



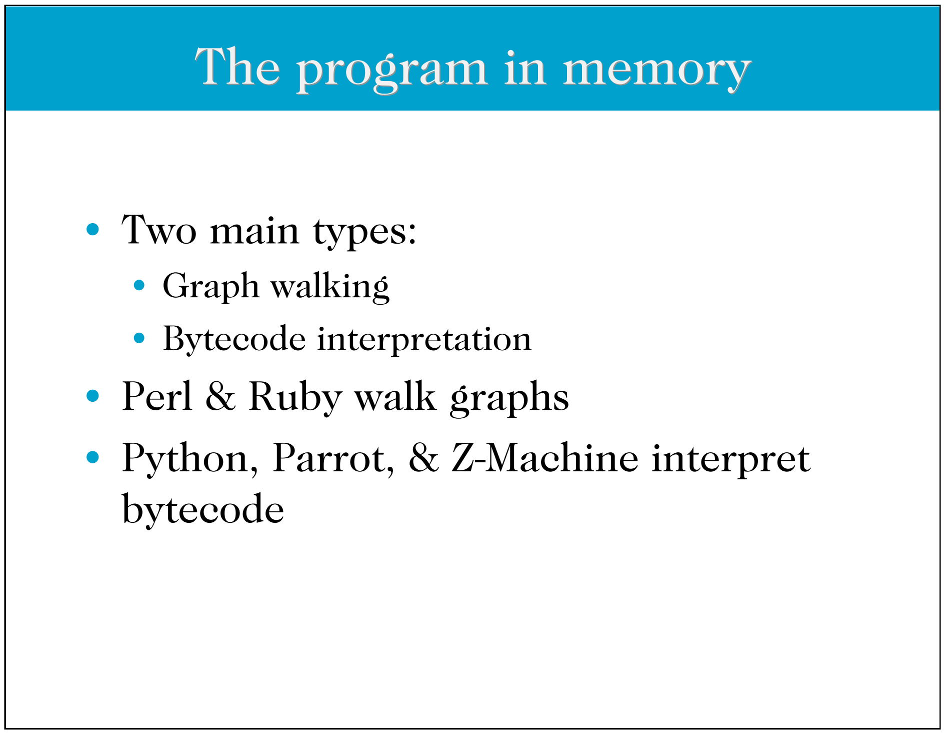
The rest of the talk in a nutshell

- Powerful text processing is your friend
- Domain-specific languages are very handy
- Writing compilers happens when you least expect it
- Make friends with perl, Parse::RecDescent, Text::Balanced, or their equivalents

Or, in other words, we preprocess the heck out of a lot of the code. Which we do. Most of the code the compiler ultimately sees has been generated by the preprocessor. (A perl-based preprocessor, not C's sad excuse for a preprocessor)

How much code that is depends on when you look, and how you count. We're at about 60K lines of programmer-editable source (including comments) and about 125K lines of compilable source. Not counting the mandatory class library code.

Anyway, the point here is that a powerful preprocessor is a good ticket to productivity and maintainability.

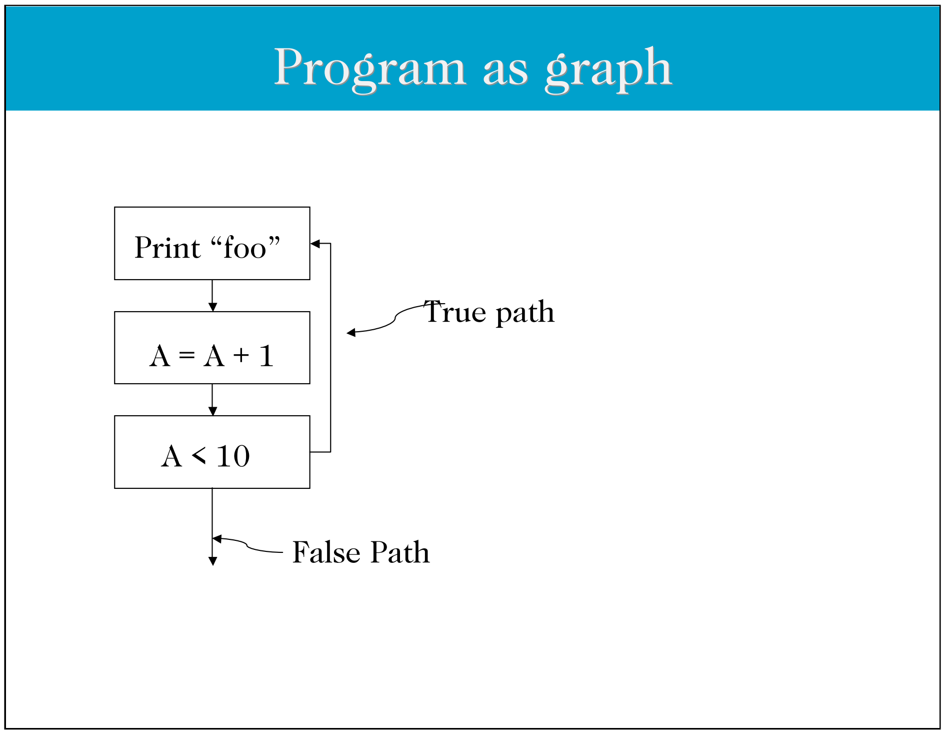


The program in memory

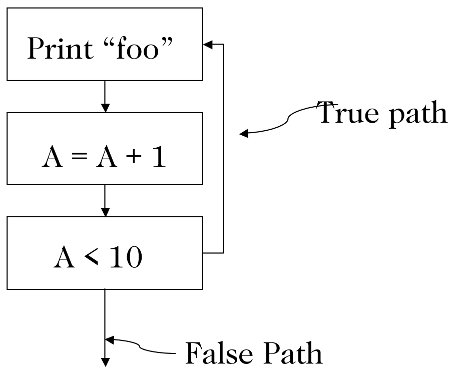
- Two main types:
  - Graph walking
  - Bytecode interpretation
- Perl & Ruby walk graphs
- Python, Parrot, & Z-Machine interpret bytecode

So here we're talking about how an interpreter can, well, interpret things. Generally the compiler part generates either a tree structure that represents the program or a bytecode stream. This isn't necessarily relevant, but it's useful to keep in mind, since the in-memory program representation does have a bearing on what the most effective means of executing it is.

Anyway, perl any ruby build up a graph in memory, while python, parrot, and the Z-machine all use a bytecode scheme.

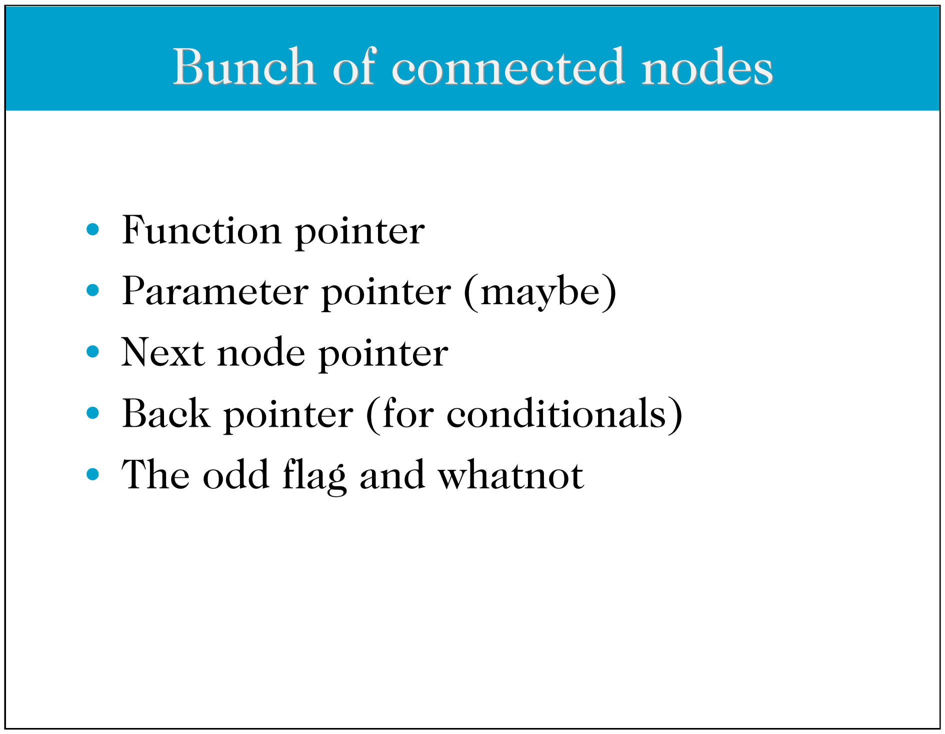


Program as graph



Yep, this is just the *tiniest* bit simplified here. Each graph node is a structure, and things like “a = a + 1” are normally a set of nodes themselves rather than a single node.

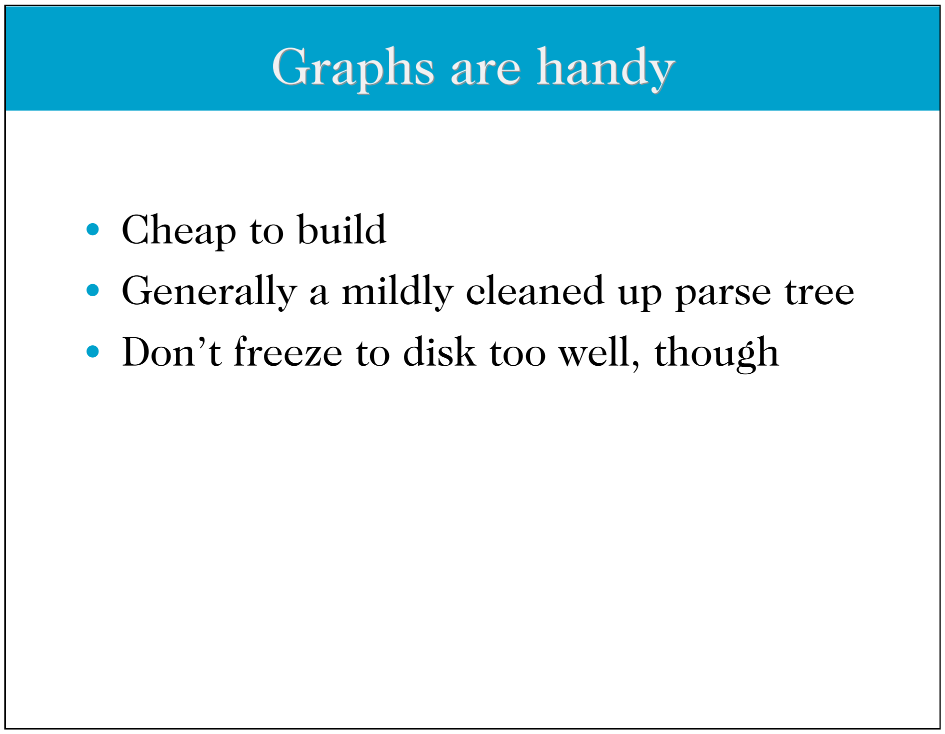
The tree nodes are usually variable size and shape, depending on their use. Decision nodes usually have a true and false pointers, nodes that do things have pointers to their parameters and the next node in the tree, and so on.



Bunch of connected nodes

- Function pointer
- Parameter pointer (maybe)
- Next node pointer
- Back pointer (for conditionals)
- The odd flag and whatnot

Most of these are optional. A next pointer of some sort and a function pointer are the minimum necessary for a node (since you need to know what to do and where to go next) with the rest optional. Most systems have variable-sized nodes so they only take up as much space as needed, though the code's often a bit messier because of it. (You get into unions of node structs, node pointer casts, and stuff like that)

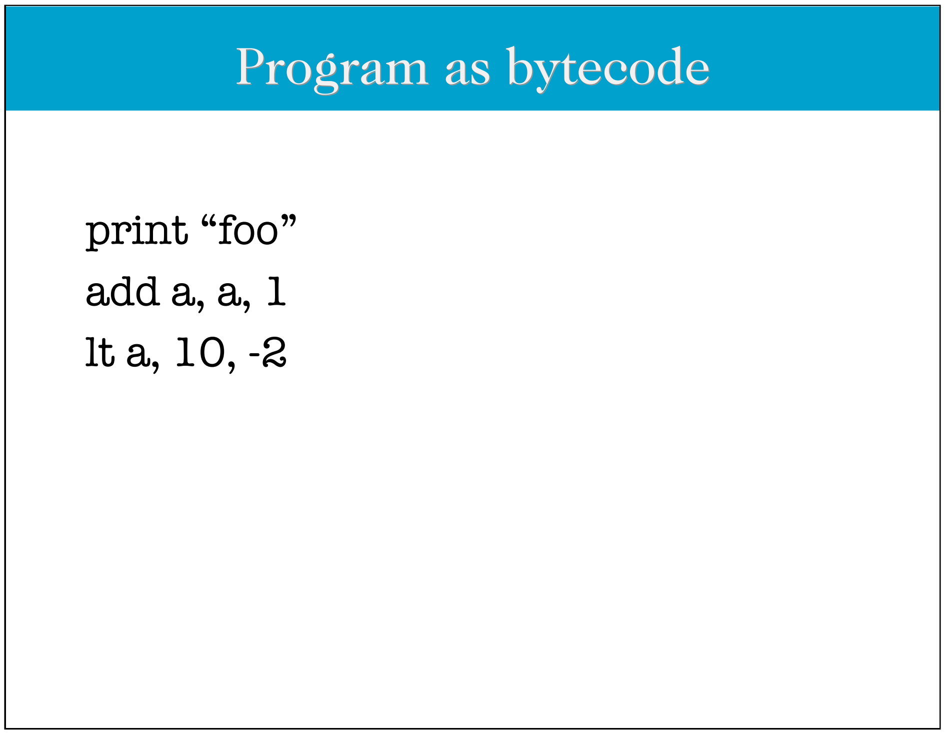


Graphs are handy

- Cheap to build
- Generally a mildly cleaned up parse tree
- Don’t freeze to disk too well, though

Points one and two are related here. For the graph walkers, what they generally do is build up the graph while parsing the source, then clean it up a bit and just walk it. This speeds up compilation a bit and makes things simpler internally, though it does mush the runloop, node function code, and parser code all together, at least conceptually. That makes it somewhat difficult to change how one part of the system works without breaking all the other parts. On the other hand, since nobody really does, this isn’t a huge issue. (Why they don’t is an open question)

The final point is interesting. Perl 5, for example, has a bytecode freeze/thaw system, which’ll take a program and freeze it to disk as bytecode, or load bytecode from disk and thaw it out for execution. Turns out to be rather slower than just recompiling the program from source--the extra IO (the bytecode is significantly larger than the source) and processing needed to turn the bytecode back into a tree take more time than compiling fresh. Go figure.

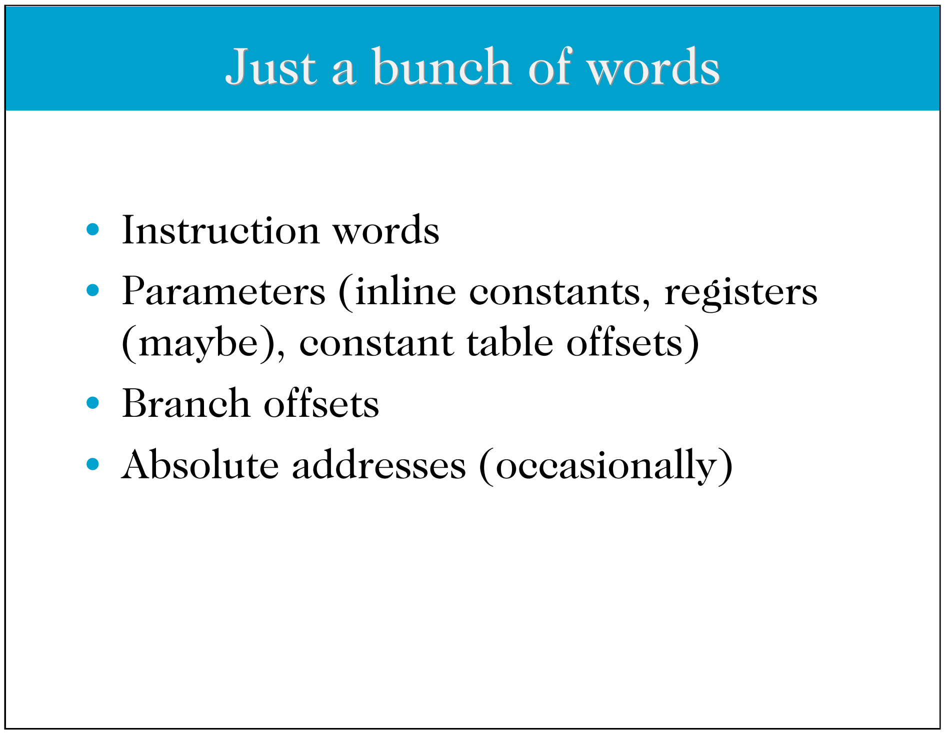


Program as bytecode

```
print “foo”
add a, a, 1
lt a, 10, -2
```

Yep, the standard “opname, destination, source…” format. Viva le VAX!

This is also assembly code rather than bytecode, if for no other reason than a hex dump is difficult to decipher on-screen.

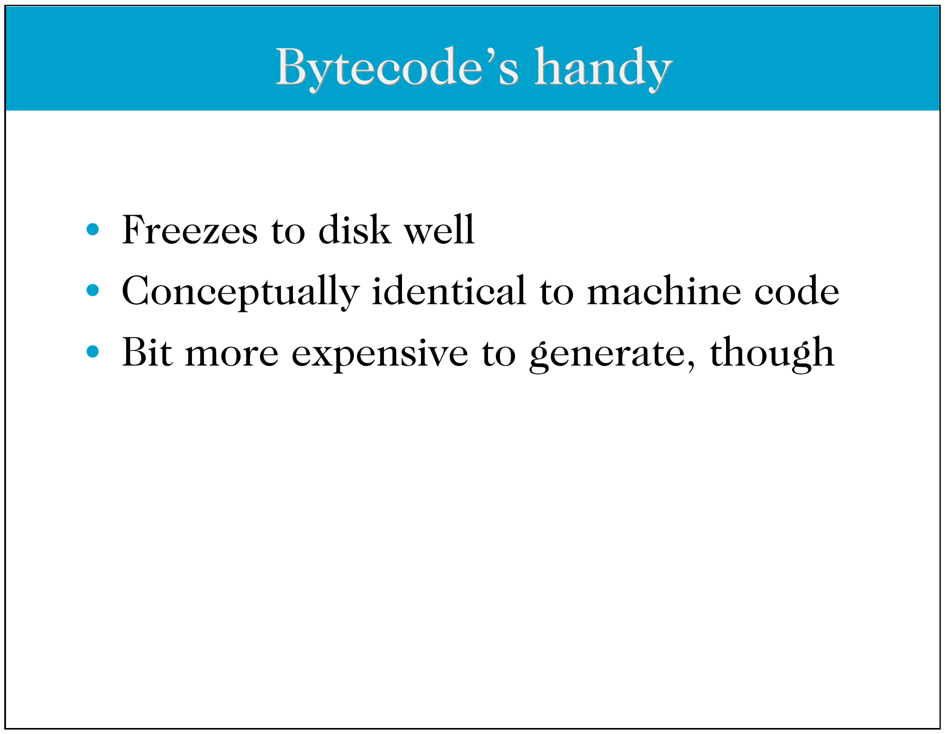


Just a bunch of words

- Instruction words
- Parameters (inline constants, registers (maybe), constant table offsets)
- Branch offsets
- Absolute addresses (occasionally)

Words here are machine words. Possibly bytes, more likely 16 or 32 bit quantities. Oddly enough, not 64 bit quantites, even on 64 bit machines. (The efficiency gain from using native sized words is offset by the time it takes to get the extra data from memory. It's possible that at some point memory busses will again be fast enough that this isn't the case)

Like any other machine code, which is all bytecode really is, there are inline constants and constant table offsets, since pure code with no data is somewhat… limited in what it can do without being extraordinarily clever. There's no obligation for the inline constants and table offsets to be the same size as an instruction word, but it does make things easier if they are.



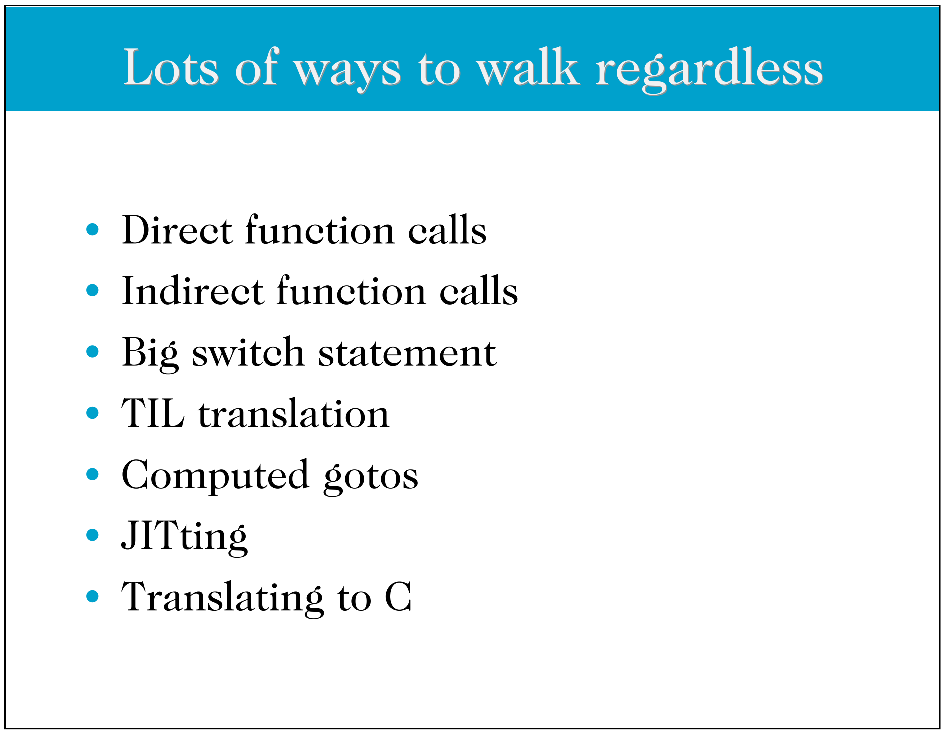
Bytecode's handy

- Freezes to disk well
- Conceptually identical to machine code
- Bit more expensive to generate, though

Well, freezes to disk well if you're paying attention when you design the bytecode. It's certainly possible to really screw things up here.

Something that's very important to keep in mind is that you really, really, really want your bytecode to be read-only. If you have to process your bytecode when you load it then you've made a really big mistake. (Note that this doesn't take into account things like constant tables, which likely require processing unless you're lucky enough to be able to get a hold of the hardware MMU, in which case things become phenomenally easier. And far more dangerous, but there are days I'd really, really love to be able to do that…)

Bytecode takes more work to create than a tree does, mainly because in almost all cases you get the tree for free as part of the parsing of the source, and then have to process it to generate the bytecode. The generation time's generally pretty trivial, though. (Not free, however, so worth keeping in mind)

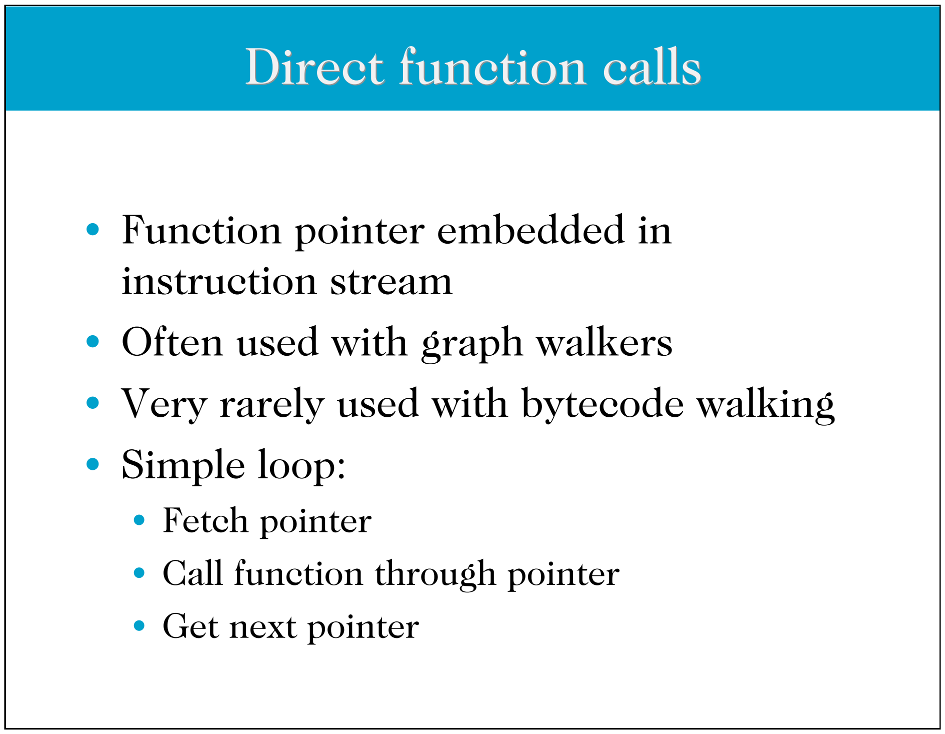


Lots of ways to walk regardless

- Direct function calls
- Indirect function calls
- Big switch statement
- TIL translation
- Computed gotos
- JITting
- Translating to C

Yes, the big list o' ways to execute the code that the tree/bytecode represents, and I may well have forgotten some. Each has its own slide, so look to those slides for more details.

FWIW, for parrot we use most of these, which is detailed later too.



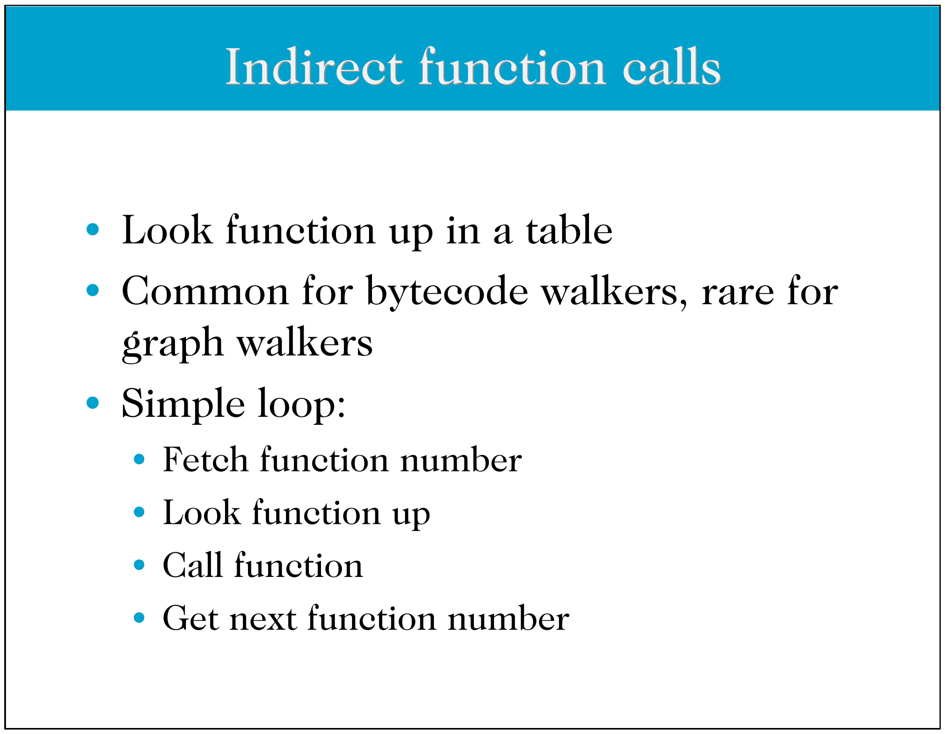
Direct function calls

- Function pointer embedded in instruction stream
- Often used with graph walkers
- Very rarely used with bytecode walking
- Simple loop:
  - Fetch pointer
  - Call function through pointer
  - Get next pointer

Perl 5 does this. It works out OK, though there are some issues, like it's really tough to change the function that all nodes of a particular type call without walking the tree. That's not too big a problem, since it's an uncommon thing to want to do. (Darned useful, though)

Basically each node has a pointer to the function that processes that node. When the runloop gets to the node it just yanks out the function pointer and calls into it. Straightforward enough, if you're comfortable with indirect function calls, and they're really not that bad. (Well, they're a pain in the neck syntactically in C, but that's hardly a ringing condemnation)

Bytecode engines normally don't do this--to do so you'd likely have to preprocess the bytecode and replace the opcodes with the function pointer addresses, which is a lot of preprocessing. Doable, though you're halfway to TIL at that point, so you might as well go all the way.



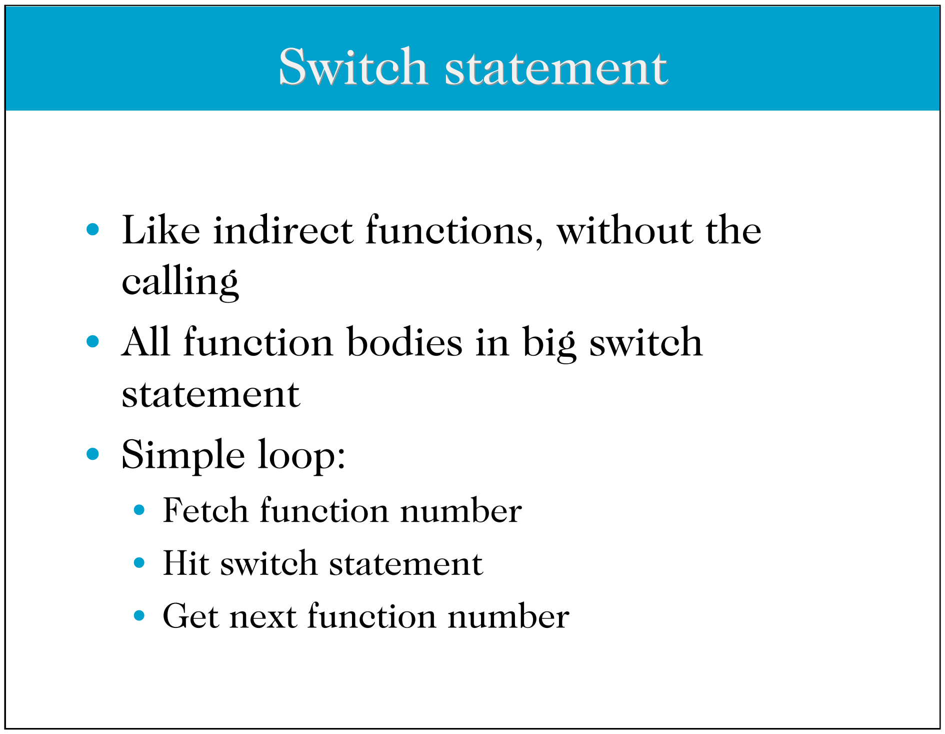
Indirect function calls

- Look function up in a table
- Common for bytecode walkers, rare for graph walkers
- Simple loop:
  - Fetch function number
  - Look function up
  - Call function
  - Get next function number

In this case, rather than have a function pointer, you've a function number, which you look up in a table. Bytecode runloops usually do this, because all you have with the bytecode is a function number. Tree walkers have a more complex structure to represent an action, so they can stuff more things in there.

It's worth noting that there is a performance penalty here, since you've got an extra level of indirection, for optree walkers. (You look up the node, then look up the function, then call) I taught perl 5 how do do this once, so we could override opcode functions at runtime. It added a nasty penalty (10% sticks in my mind, though that could be wrong) to the runtime, and just wasn't worth it.

Parrot has a core that works this way. It's the slowest core, though the overridability's useful in some cases. It's more useful in a mixed-core, where most functions are *not* called this way, but some are, since you can then have a set of fast, known functions and everything else looked up at runtime. Not only can you override functions this way, you can also add functions at runtime. Basically you've got a CPU with loadable microcode.



Switch statement

- Like indirect functions, without the calling
- All function bodies in big switch statement
- Simple loop:
  - Fetch function number
  - Hit switch statement
  - Get next function number

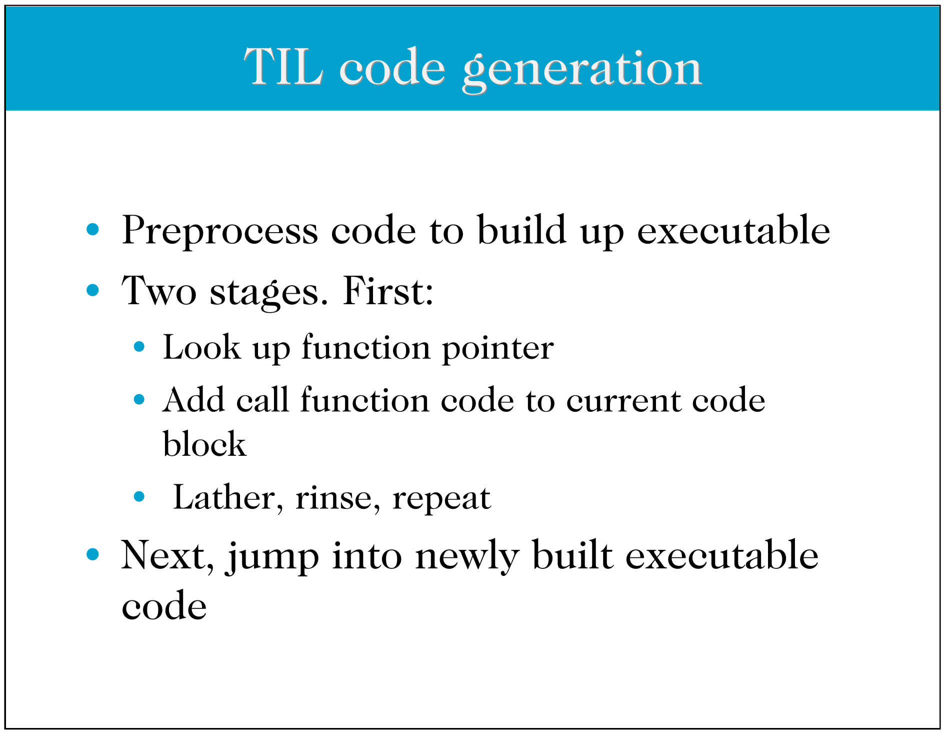
Mmmm, the big switch. These are always fun, with this core being used by Python.

How efficient they are depends a lot on your C compiler and the size of your switch statement, since there are a number of different ways to generate code for the switch. Regardless, the big win you get here is that, even if the code generated by the C compiler to determine which label to go to really stinks, you at least skip executing the compiler's function preamble and postamble code.

(The preamble code is the code that the compiler generates for a function that gets executed when the function is entered, before the actual function code. It's setup code which does whatever housekeeping's necessary. The postamble code is the code that gets executed after you leave a function, usually more housekeeping code)

How much of a win this is depends a lot on how expensive the function preamble/postamble code is. That stuff's usually pretty efficient, since it has so much impact on the speed of the compiled code, but it's also limited in what it can do by the platform calling conventions.

Regardless, since not doing something is faster than doing something, skipping that code is usually a win. (Though *not* always, since some c compilers emit code that's utter crap for switch statements in some cases--giant if ladders and the like. That's rarer than it used to be, luckily)



TIL code generation

- Preprocess code to build up executable
- Two stages. First:
  - Look up function pointer
  - Add call function code to current code block
  - Lather, rinse, repeat
- Next, jump into newly built executable code

TIL, or Threaded In Line code. No, there are no threads in the multiprocessing sense here. Instead what happens is that the runtime engine builds up a chunk of executable code by running through the bytecode or optree and emitting function call code, then executing the generated code.

So if you had bytecode like:

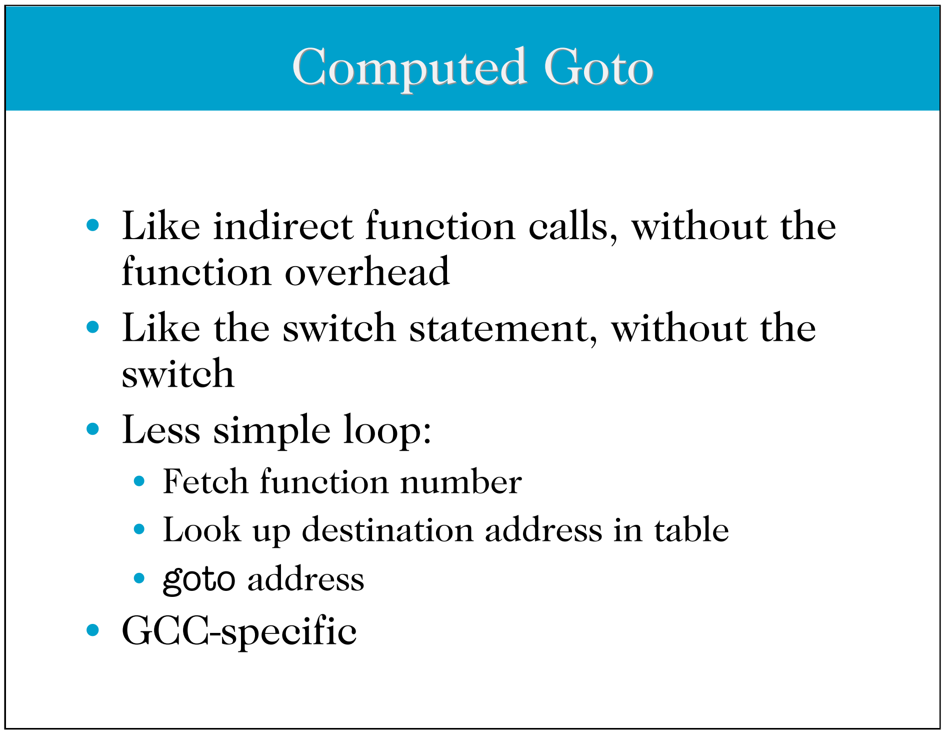
```
add I1, I2, I3
sub I1, I1, I5
```

The TIL code would be the machine level equivalent of:

```
add(I1, I2, I3);
sub(I1, I1, I5);
```

It's sort of a primitive JIT, where all you need to know how to do is build up function call code. It's actually a good place to start if you're going to build a general-purpose JIT--you start with emitting function calls and slowly start substituting in more specific code for each op you work with.

The win here is that there's no runloop, indirection, or anything in the final code. Indirection's a performance killer, since it flushes CPU pipelines, confounds the prefetch hardware, and generally makes life hell for hardware designers. With TIL the hardware knows where it's going so it can prefetch, do predictive execution, and all sorts of other things that makes your code run faster, and fast's nice.



Computed Goto

- Like indirect function calls, without the function overhead
- Like the switch statement, without the switch
- Less simple loop:
  - Fetch function number
  - Look up destination address in table
  - goto address
- GCC-specific

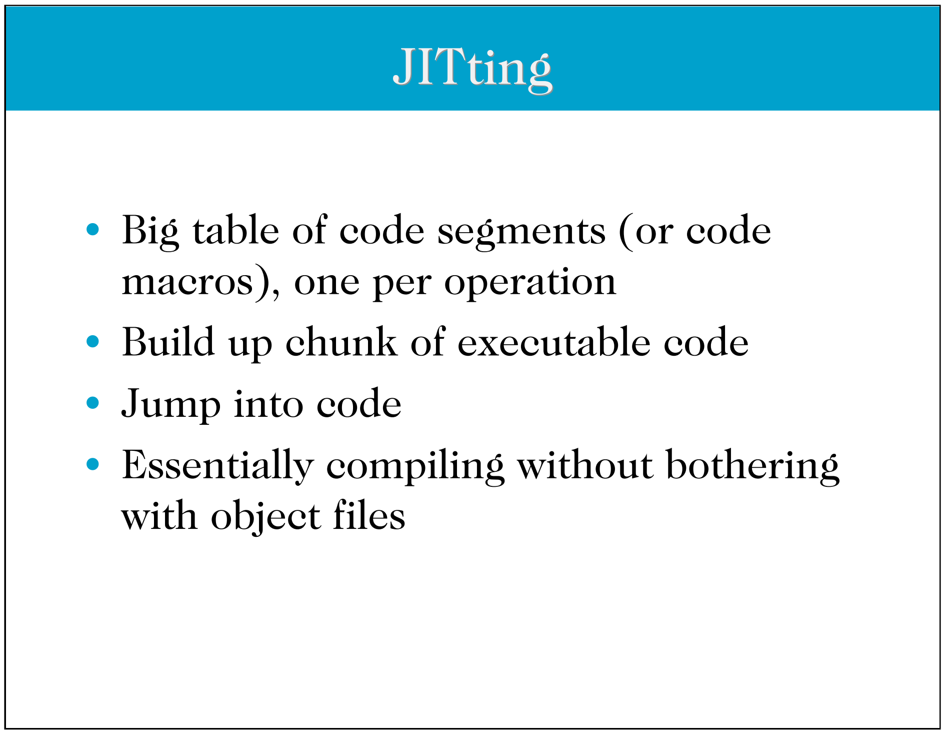
This is a compiler-specific optimization, which GCC supports and some other compilers have added in. (I'm pretty sure it smacks the optimizer so I can understand not putting it in)

Computed goto is an extension of C's goto statement. In standard C, you can only goto a label by name. With computed goto, you can goto a label indirectly--that is, rather than using a label as a destination you use a variable, and the value of the variable is taken to be the destination address. There's an extra addition to C where you can take the address of a label. Between the two, you can jump to any arbitrary piece of code you want to. It's not entirely safe--because the compiler can't tell where you're going it's possible to get your stack messed up really badly.

With this, though, you put all the opcode function bodies in one big function, with each opcode body having a label attached to it. You build a table of function pointers, indexed by opcode number, then have each opcode function, when it's done, do a goto to the function body of the next op.

Computed goto cores tend to really annoy GCC's optimizer, even worse than the switch core does. Don't be surprised when it takes up huge wads of memory (more than 192M for Parrot's computed goto core) and then bails because your code's to complex to optimize anyway.

In our experience, this is the fastest non-JIT core, and can be faster than a partial-JIT core. (That is, a core where some opcodes are JITted while the rest have TIL-style 'call this function' code inserted for them)



JITting

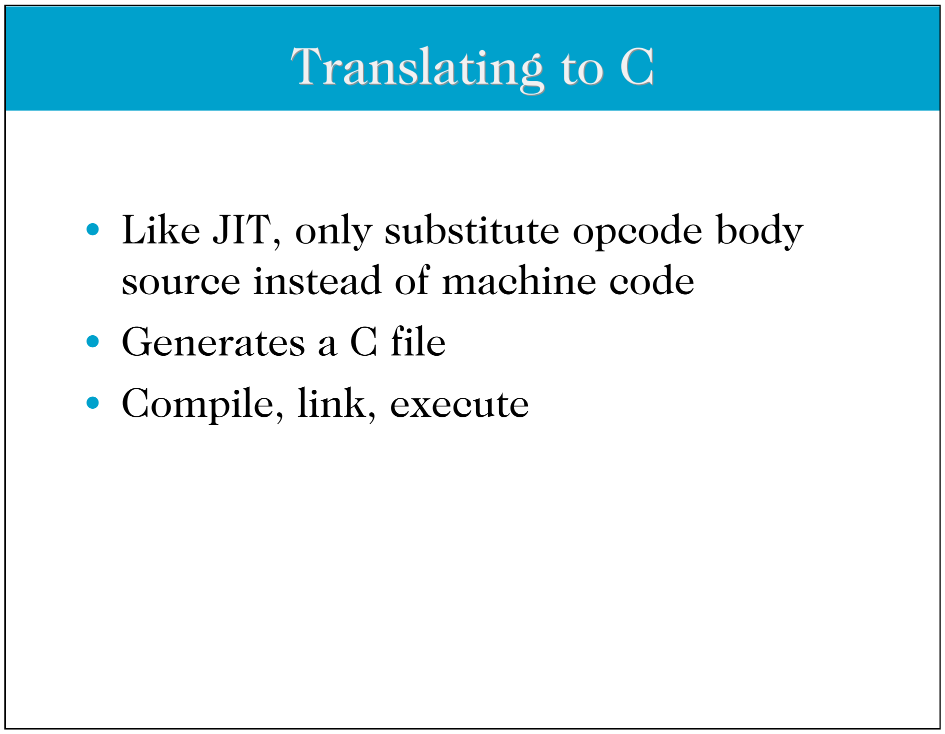
- Big table of code segments (or code macros), one per operation
- Build up chunk of executable code
- Jump into code
- Essentially compiling without bothering with object files

A JIT is, for all intents and purposes, a simple compiler. Since the JITting is done on the fly, the compilation's normally pretty simple, without much optimization. That's in large part because many optimizations can take an awful lot of time, which you don't want to spend since you're just going to throw away the optimized code when the program exits anyway. (Also some bytecode and optree formats have little enough information that optimization is difficult)

JITs are often thought of as bizarre, arcane things, possibly because to start only bizarre and arcane people did them. (That most of those folks were working on Java JVMs and likely horribly scarred for life because of that didn't help, I'm sure) Turns out that they're really not that big a deal once you get past the whole "JITs are Hard!" idea. They really aren't.

What they are, though, is a pain, since the emitted code is custom to each processor family, and sometimes to each processor in the family, or operating system running on the processor. But not really tough.

Parrot's got a JIT, though we do have a few unusual issues to deal with. The large instruction set's one of them, and the possibility of runtime-loadable opcodes are another. We deal with 'em though--the JIT knows how to generate TIL-style function call code on every platform it runs on, and we poke in opcode functions one by one as we go. Loadable opcode functions just get called TIL-style, though we really ought to spec out the information we need so that loadable opcode libraries can provide JIT data too.



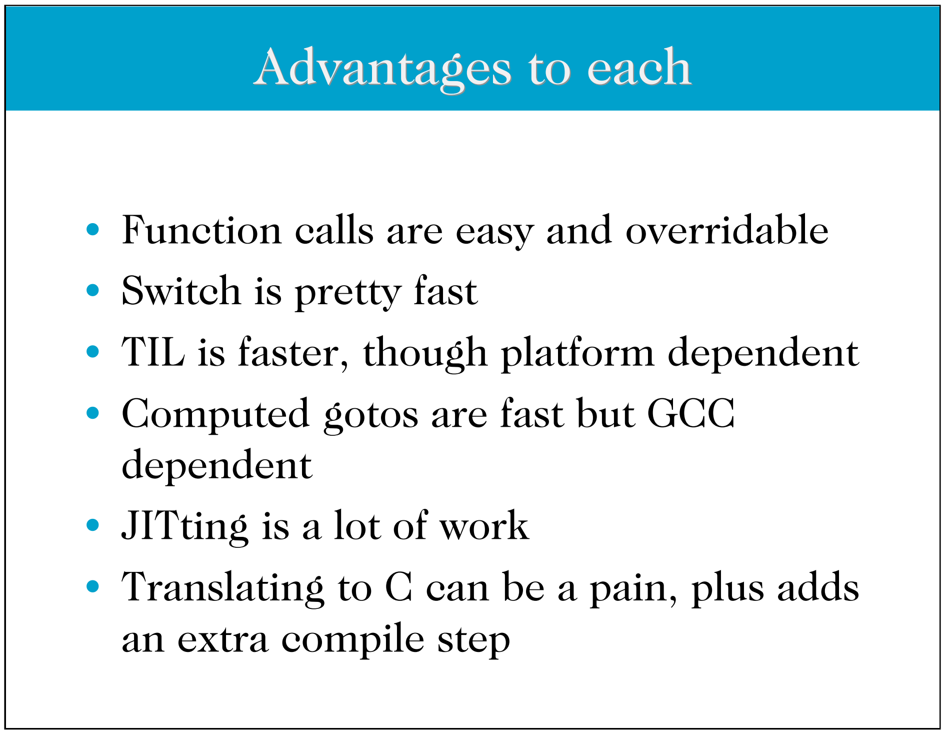
Translating to C

- Like JIT, only substitute opcode body source instead of machine code
- Generates a C file
- Compile, link, execute

And our last step… C! Or some other compiled language, but C’s pretty common. (Which is a pity. Fortran would be better in those cases where its feature set suffices, but, well… there you go)

In this case, rather than executing the bytecode in any way, the engine translates it to C code and hands it off to the C compiler. (Well, it may just save the source on disk) Then, rather than building your own optimizer, you just use the C compiler’s optimizer. If your VM is written in C then you can probably play some other tricks (which Parrot dies) to make generating the code really easy. You could, if you wanted, not even bother having a runloop or anything of the sort for your VM, and just generate C code, compile it, and dynamically link it in when bytecode’s loaded or generated.

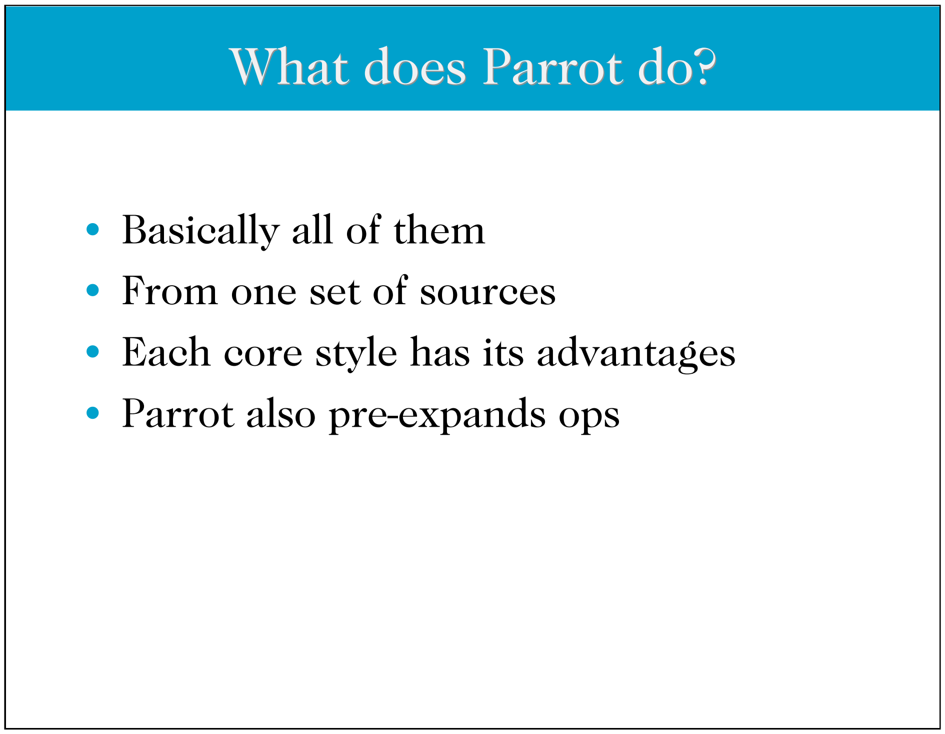
This also has the bonus feature of generating a real platform object files to link into other programs, which is kinda nice.



Advantages to each

- Function calls are easy and overridable
- Switch is pretty fast
- TIL is faster, though platform dependent
- Computed gotos are fast but GCC dependent
- JITting is a lot of work
- Translating to C can be a pain, plus adds an extra compile step

Tradeoffs, tradeoffs, tradeoffs. Each of the cores, even the ones you'd think were horribly slow, all have their own advantages, and which one is 'better' depends on the circumstances. Even within a single program run you may want one or another core depending on what the code's doing.



What does Parrot do?

- Basically all of them
- From one set of sources
- Each core style has its advantages
- Parrot also pre-expands ops

Yep. Because nothing exceeds like excess.

We started with the indirect function core, because it's easy. The switch core's faster, and the computed goto core's faster still if the compiler supports it. Finally it's the JIT core, which defaults to TIL-style function call generation for ops that the JIT has no knowledge of.

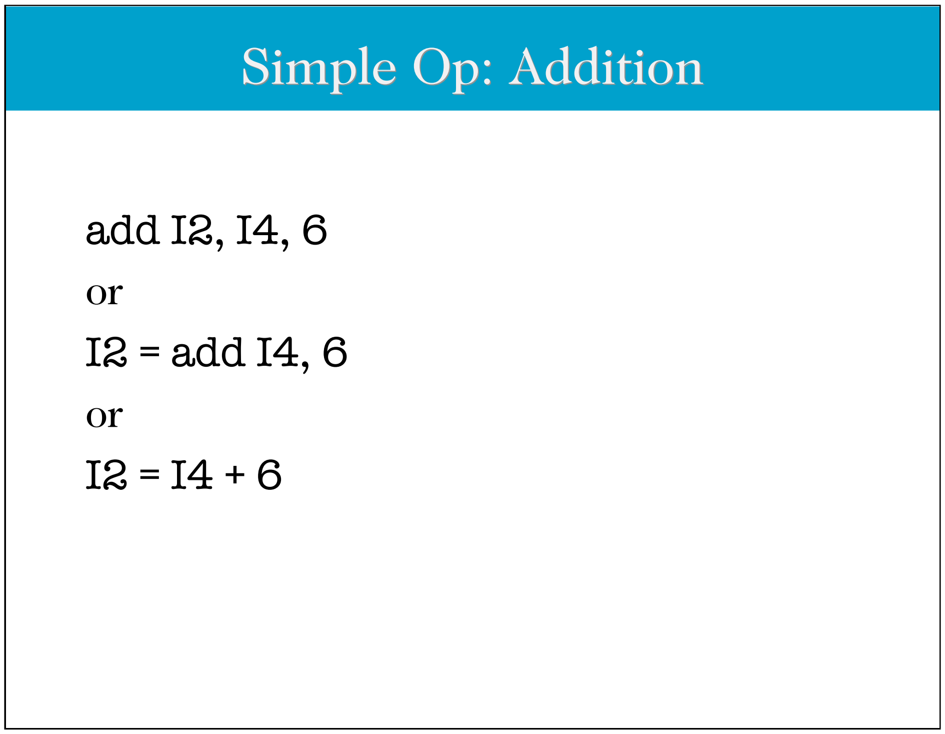
The preexpanding part is how we do CPU style stuff without paying the runtime cost of argument decoding. With hardware, the type of the argument is encoded in the argument, and there's hardware dedicated to figuring out whether it's a constant, a register, an address, an indirect address, or any of a half-dozen other things.

With a setup like that you have a single add opcode, with the source and destination types encoded in the arguments. We *could* do that with parrot, but it'd mean a lot of processing for each opcode. That's silly, especially since we know at compile time what the arguments are, so we can encode that in a way that's more easily used. So…

What we do is, rather than have one add op that can take a dozen or more types of parameters, we have a dozen add ops which each take a single type of parameter. This means there's no runtime checking--when we hit the "add an integer register and an integer constant and stick it in an integer register" op, well, we do just that. No decisions, no checks, nothing but op function.

While this does make for more code, today's processors have more cache so there's space for more specialized code, and generally a program will favor a subset of the possible functions so it's less of a hit than you might expect, or so we hope at least.

This also makes the JIT easier to implement. More tedious, mind, but easier--since each individual op is much simpler to do, so folks can get into doing JIT work with less hassle. That's the theory, at least.



Simple Op: Addition

```
add I2, I4, 6
or
I2 = add I4, 6
or
I2 = I4 + 6
```

Yep, there are three acceptable syntaxes here. The first is plain old parrot assembly, which is also acceptable PIR. (PIR being Parrot Intermediate Representation, a step up from assembly) The third form is PIR's infix notation, which supports the basic math operations.

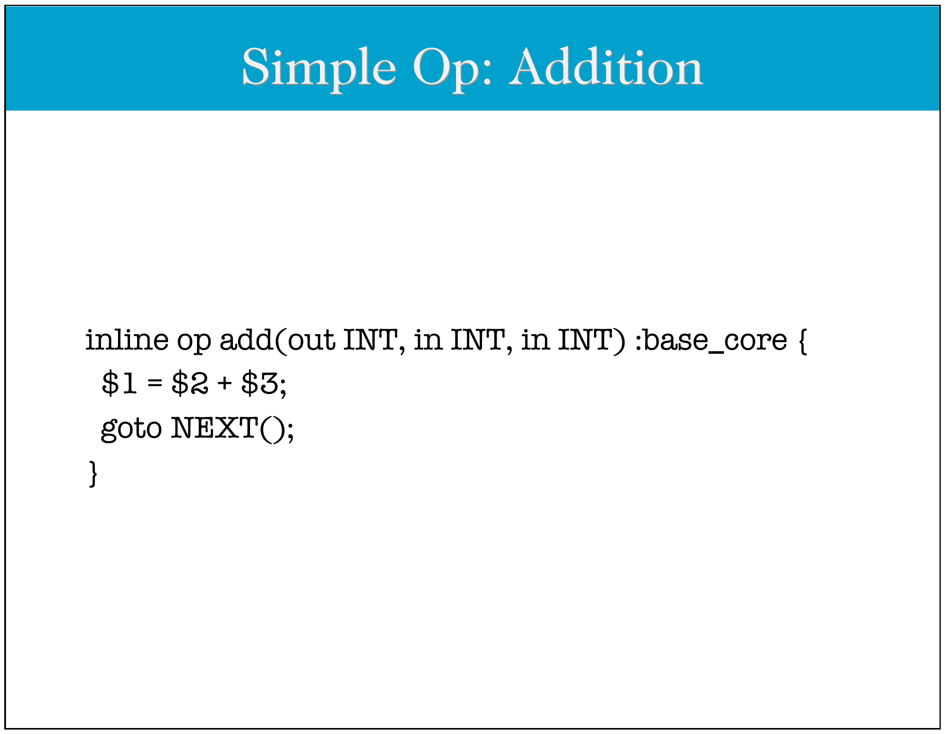
The second form's a clever transform thing that takes advantage of the fact that with parrot ops the destination is always *first*. The PIR compiler generically turns everything in the form:

```
x = opname y, z[, aa…]
```

Into

```
opname x, y, z[, a…]
```

So in addition to the basic math ops you can use pretty much any op that provides a value in a reasonably intuitive way. Doesn't much matter when writing compilers, but it makes writing this stuff by hand nicer.



Simple Op: Addition

```
inline op add(out INT, in INT, in INT) :base_core {
  $1 = $2 + $3;
  goto NEXT();
}
```

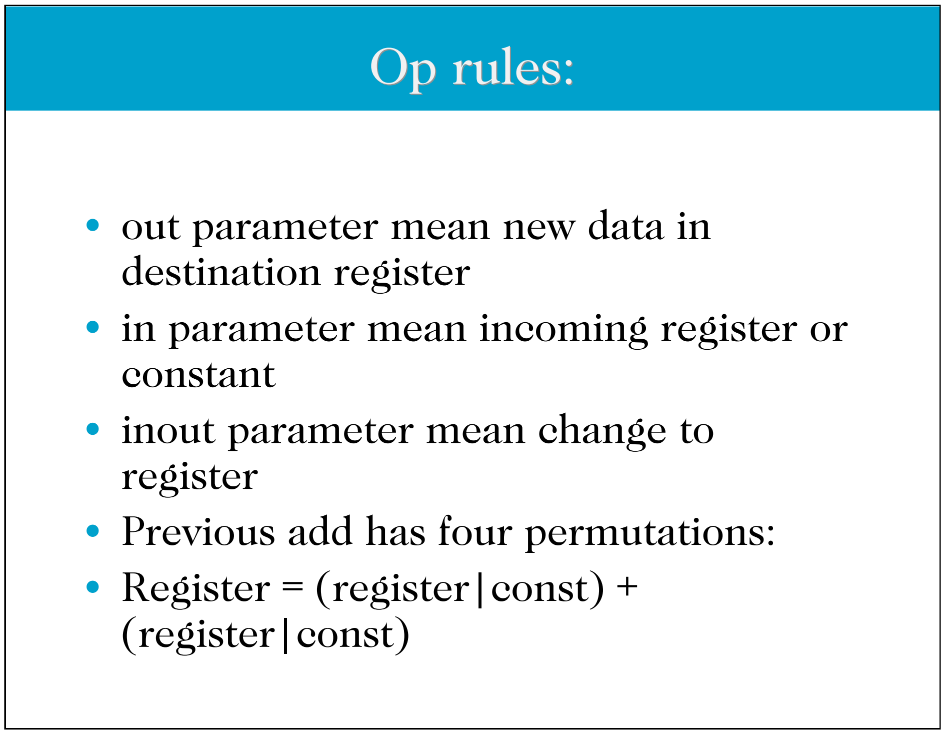
This is the source for the add opcodes which take an integer register or constant, and return the value in an integer register. All four of them. No, the source isn’t duplicated, we preprocess this into what the C compiler needs. Makes maintenance nice.

The “in INT” tells the preprocessor that this can take either an integer constant or an integer register. The “out INT” tells the preprocessor that this returns an integer value, which means it has to be a register, and it’s also a hint that the destination is changed, though we don’t use that right now. We might someday.

The one downside is that we do duplicate this function for NUM, PMC, or STRING parameters, which is kind of a bummer, but for most ops this isn’t a big deal as we need to do very different things to PMCs or STRINGs than we need to do to INTs.

The :base_core thing marks this op as part of the basic core ops. We can tag ops with attributes which we may or may not ever need.

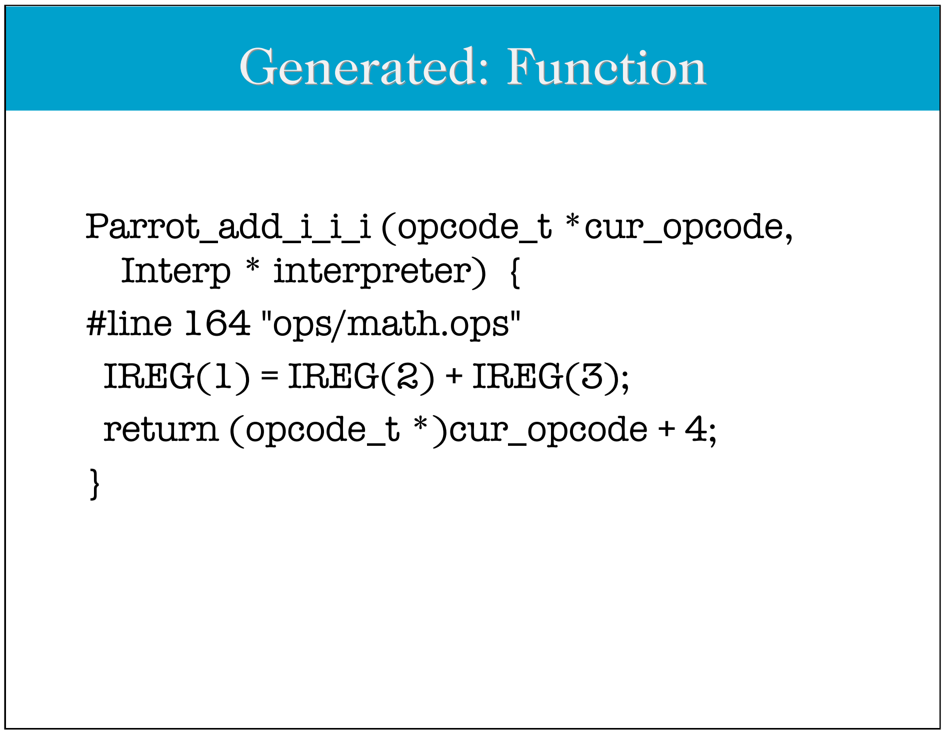
Finally that “goto NEXT();” thing isn’t a C goto. It’s a magic token telling the preprocessor that at this point in the op function we can go and dispatch to the next op in the stream.



Op rules:

- out parameter mean new data in destination register
- in parameter mean incoming register or constant
- inout parameter mean change to register
- Previous add has four permutations:
- Register = (register|const) + (register|const)

Yep, the notes for the previous slide explained all this.



Generated: Function

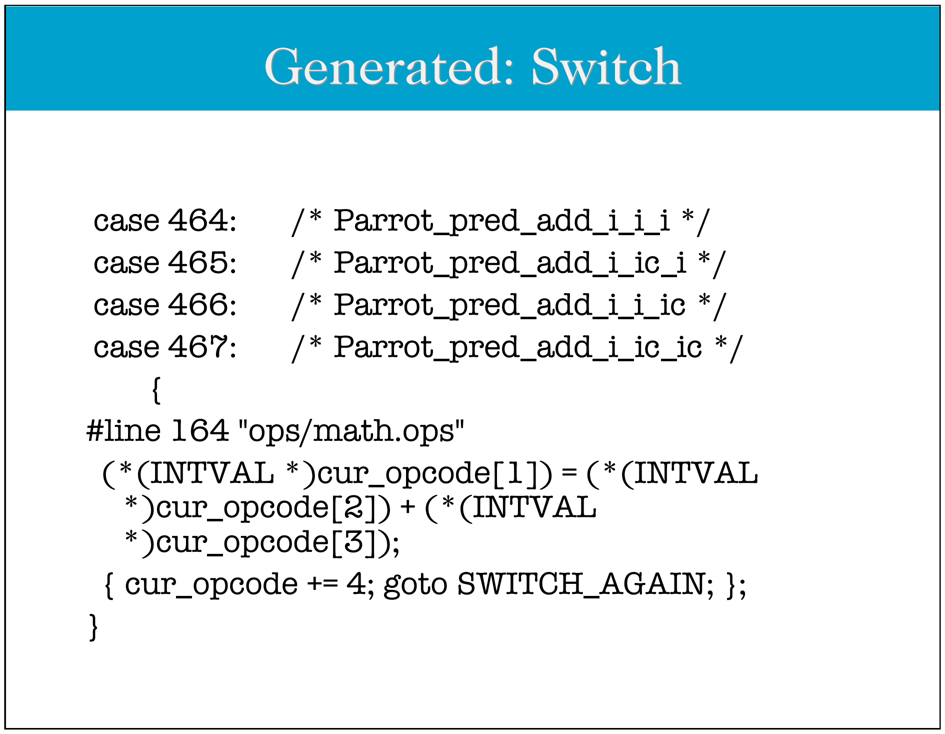
```
Parrot_add_i_i_i (opcode_t *cur_opcode,
  Interp * interpreter) {
#line 164 "ops/math.ops"
 IREG(1) = IREG(2) + IREG(3);
 return (opcode_t *)cur_opcode + 4;
}
```

Isn't autogenerated C code so wonderful?

We prepend the opcode function, and all functions we may expose, with Parrot_ so as not to pollute the OS-level namespace. (Which is a problem we always run into linking in libraries. It's amazing how many libraries expose a function named handle_error and suchlike things)

The #line thing's a C preprocessor directive to hopefully give sane error messages when things go wrong. And things do, always, go wrong. This one notes that this function comes from line 164 of the file ops/math.ops. (While all the op functions get mashed together into a single file for the compiler to see, people need to know which source module this all came from)

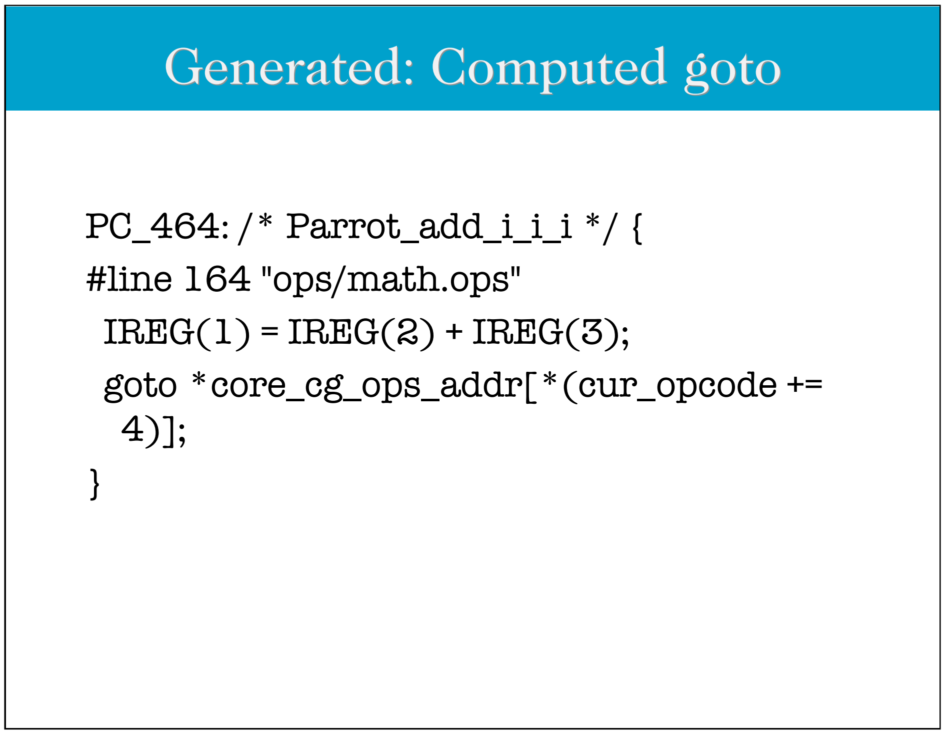
This is the indirect function version of the opcode function. There are others, which we'll get to in the next slides.



Generated: Switch

```
case 464:     /* Parrot_pred_add_i_i_i */
case 465:     /* Parrot_pred_add_i_ic_i */
case 466:     /* Parrot_pred_add_i_i_ic */
case 467:     /* Parrot_pred_add_i_ic_ic */
    {
#line 164 "ops/math.ops"
 (*(INTVAL *)cur_opcode[1]) = (*(INTVAL
   *)cur_opcode[2]) + (*(INTVAL
   *)cur_opcode[3]);
 { cur_opcode += 4; goto SWITCH_AGAIN; };
}
```

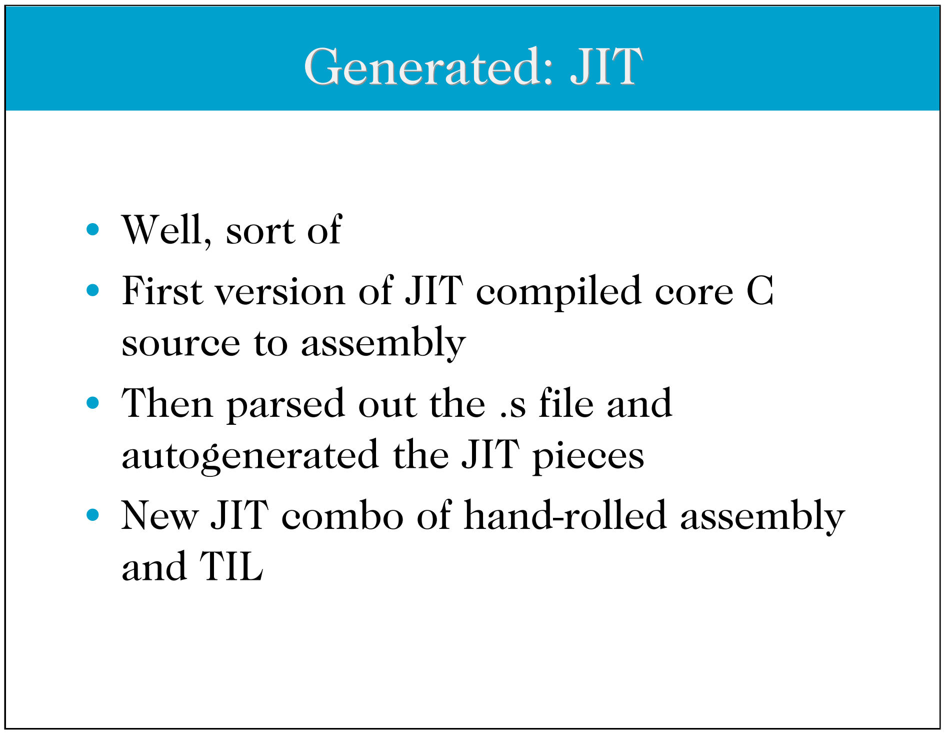
This is the code generated for our add function which goes into the big switch statement. Yeah, it’s a little weird, but that’s because of some of the games we play. Does all work out.



Generated: Computed goto

```
PC_464: /* Parrot_add_i_i_i */ {
#line 164 "ops/math.ops"
 IREG(1) = IREG(2) + IREG(3);
 goto *core_cg_ops_addr[*(cur_opcode +=
  4)];
}
```

And the chunk of the computed goto core. The labvl, FWIW, is PC_ and the opcode number, in this case 464.



Generated: JIT

- Well, sort of
- First version of JIT compiled core C source to assembly
- Then parsed out the .s file and autogenerated the JIT pieces
- New JIT combo of hand-rolled assembly and TIL

Parrot’s JIT really isn’t generated, at least not for the current version of the JIT. We did, at one time, preprocess the assembly listings that you can get from GCC and used those as the JIT core functions. Nifty, but it was fragile and the parsing was starting to get really nasty for the more complex opcode functions. (And we do generally want the code in the opcode function, rather than have it just call out to an external function to do the work, since then we have the overhead of a function call and might just as well TIL-in the function) Still, it’s clever and worth thinking about depending on your needs.

The new JIT is a bit more manual. It knows how to emit function call code (all our ops have the same identical function signature so that’s easy) for a platform, and from there we have a per-CPU, per-OS source file of ops that the JIT knows. (Our configuration system swaps in the right one at configure time)

Parrot’s profiling core tracks JITtability as part of its statistics set, so we can see where it’s worth spending time writing JIT functions. (So we can target the high-use non-JITted ops)

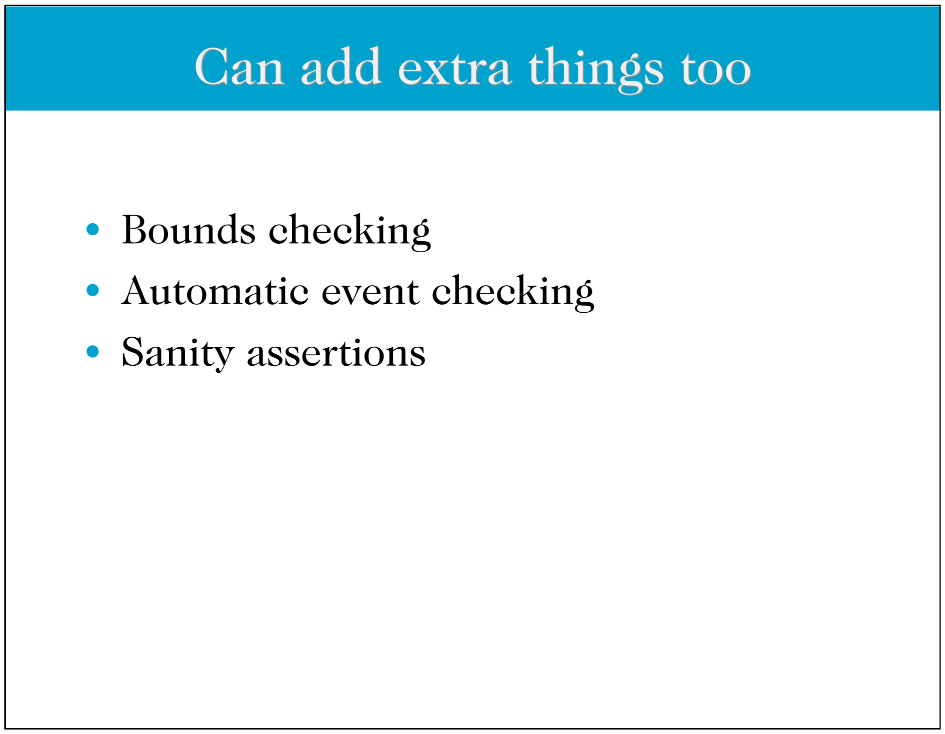
Generated: C source

```
IREG(2) = IREG(4) + 6;
```

And, finally, C source. Yeah, the IREG thing's a C macro.

Since we have to preprocess our opcode source a lot anyway, it's no big deal to also stash away the raw source into a library file somewhere and then treat the bytecode file as a sort of funky source file do macro expansion on.

From there you have a plain C file which can be compiled, linked, and run.



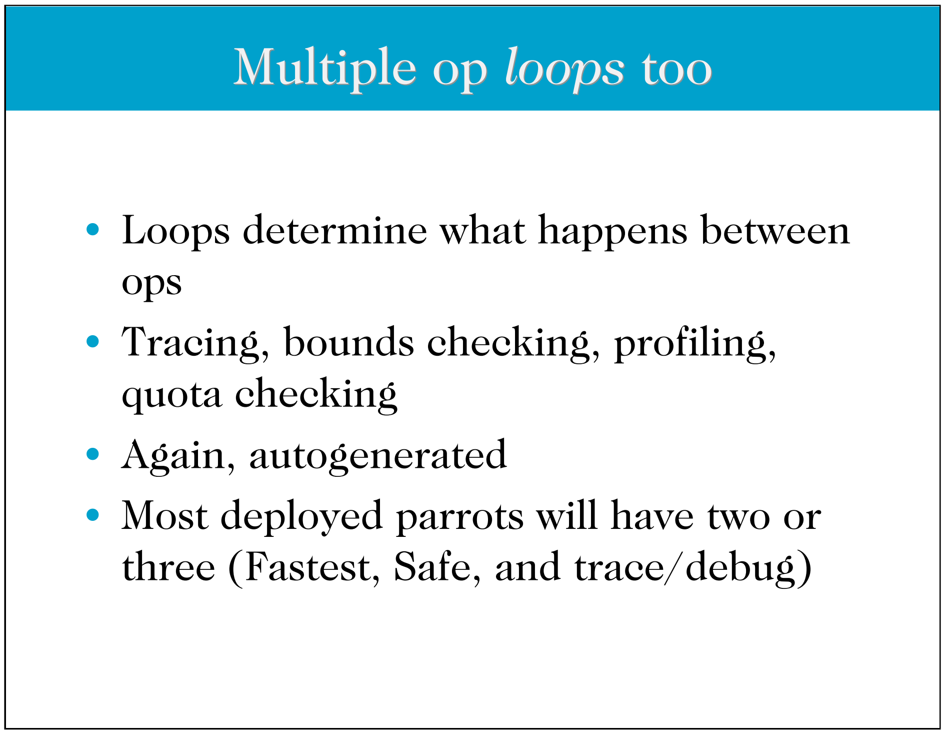
Can add extra things too

- Bounds checking
- Automatic event checking
- Sanity assertions

Since we're preprocessing the source, and have a number of well-known macros to access state, we can insert any sort of code that we might like to generate alternate cores.

For example, the normal core assumes that the register numbers are correct. This is a sensible thing to do in a case where you trust your compiler, but a really really bad idea if you don't. (Because, for example, you're running bytecode from some untrusted source, like the network) You don't want to burn the time to check code that you can assume is safe, so the normal op functions don't check.

When running unsafe code you don't want to assume, and it's OK to burn the time because it's better than getting hacked. (And there are issues with static analysis of bytecode files which we don't want to deal with) The sane thing to do then is to generate alternate opcode functions and build a separate, slower core with the security features you need.

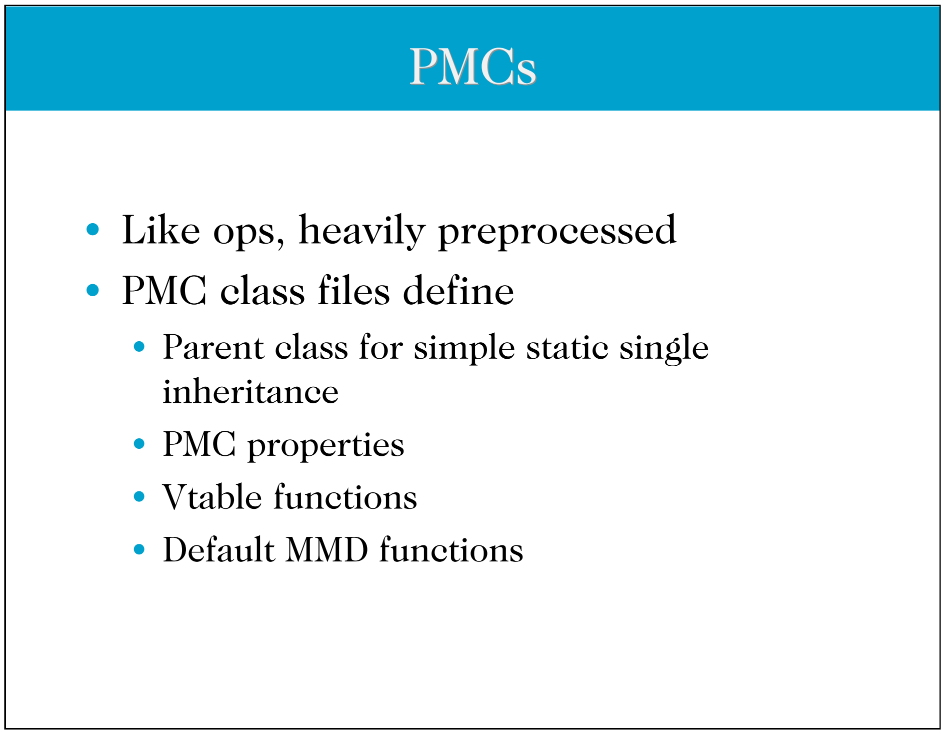


Multiple op *loops* too

- Loops determine what happens between ops
- Tracing, bounds checking, profiling, quota checking
- Again, autogenerated
- Most deployed parrots will have two or three (Fastest, Safe, and trace/debug)

In addition to messing with the op functions themselves, you can *also* have alternative runloops. This lets you do interesting things, especially if you allow for on-the-fly runloop changing. (To drop into a debugger remotely, for example)

While you want the absolute fastest runloop in the normal case, there are times where a slower but more flexible, paranoid, or functional runloop are in order.



PMCs

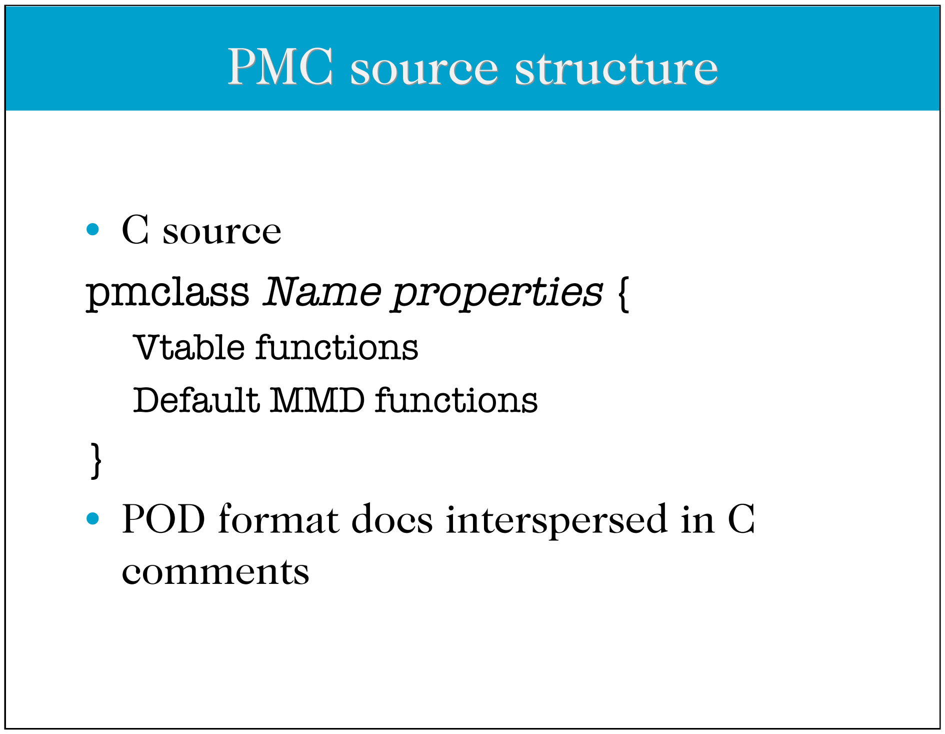
- Like ops, heavily preprocessed
- PMC class files define
  - Parent class for simple static single inheritance
  - PMC properties
  - Vtable functions
  - Default MMD functions

Ah, the lowly PMC. Short for Parrot Magic Cooke, these things represent language-level variables. Yes, they're really overkill for many languages, but remember that parrot's targeting Perl, Python, and Ruby, and languages like them, where even the basic variable has a huge wad of functionality behind them.

Anyway, each language type has a corresponding PMC type. (Which, confusingly, we call a class, even though it's not really OO, and we have other things we call classes too) Each PMC type has a set of mandatory functions it must provide, bundled up in a vtable. To make it easy to write PMC code, we heavily preprocess the source for PMCs. Oh, and do single-inheritance of vtable methods. Though they're not really classes. As such.

Naming is likely the thing we'll regret the most…

Besides the vtable, which holds only unary functions (get & set, metadata queries, and suchlike things), you can also define default MMD functions for PMC types. These are the functions that're called if there aren't any more specific functions available. Generally you'll want to register more specific functions, unless the default behaviour's OK. Which it might well be.



PMC source structure

- C source

pmclass *Name properties* {

  Vtable functions

  Default MMD functions

}

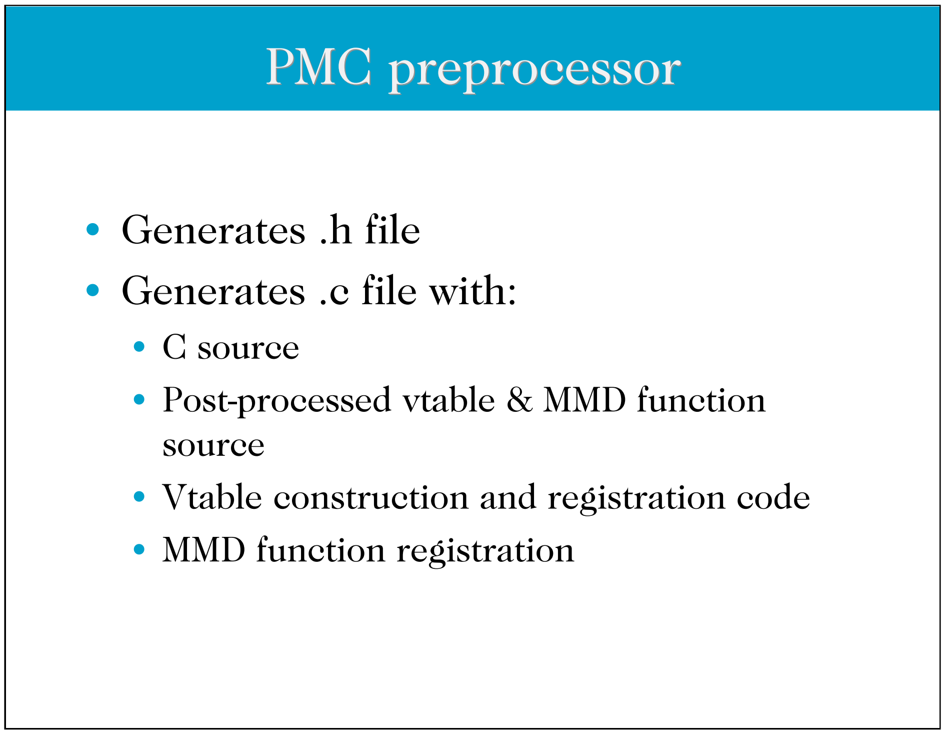
- POD format docs interspersed in C comments

Yep, it's a mixed bag. C, then a pmclass block with PMC vtable and MMD default functions defined in it. Mixed through is documentation (in C comments so it's ignored) formatted using Perl's POD formatting scheme.

Any code that appears before the pmclass block is put in the generated C file verbatim, so utility functions, #includes, and whatnot can be stuck in here.

The preprocessor has a list of function names it knows about and if those functions appear within the pmclass block their parameter list is suitably massaged (All functions have to take a self and current interpreter pointer, so we don't bother making you put them in, and we provide macros to access them) and the functions are put in the generated C file. The preprocessor also notes that they exist.

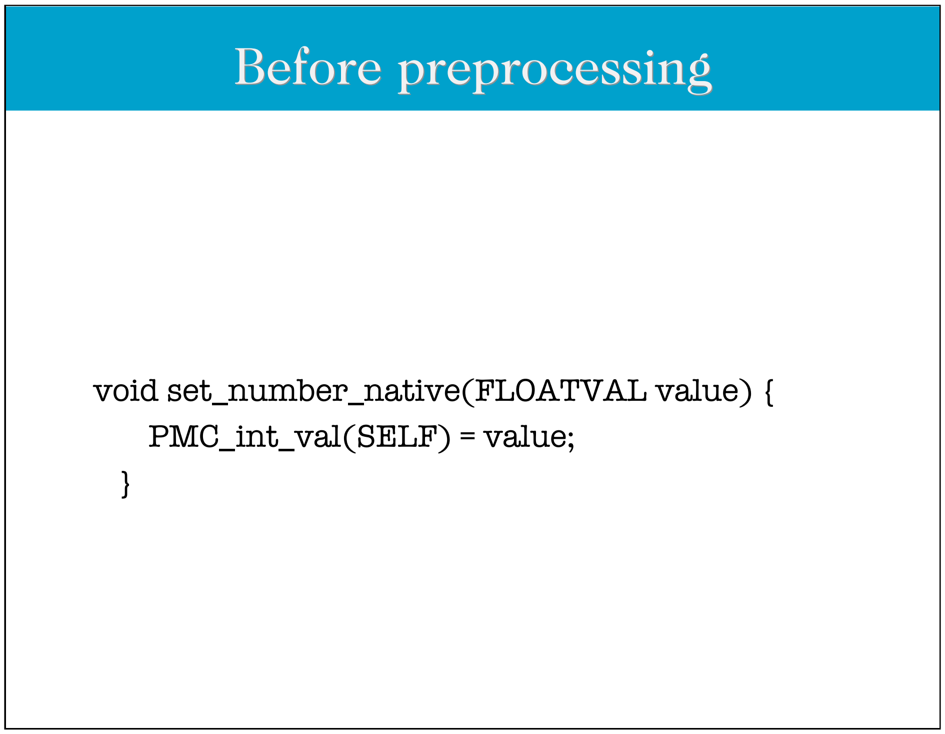
When the pmclass block has been processed the preprocessor then generates an appropriate initialization and setup code, as well as filling in any missing vtable functions with functions from the parent pmc type. (Or it's parent, back to the beginning of the tree)



PMC preprocessor

- Generates .h file
- Generates .c file with:
  - C source
  - Post-processed vtable & MMD function source
  - Vtable construction and registration code
  - MMD function registration

Like the last slide’s explanation says, we build up a bunch of stuff from the provided source, and default in whatever’s not provided. Works out pretty well.



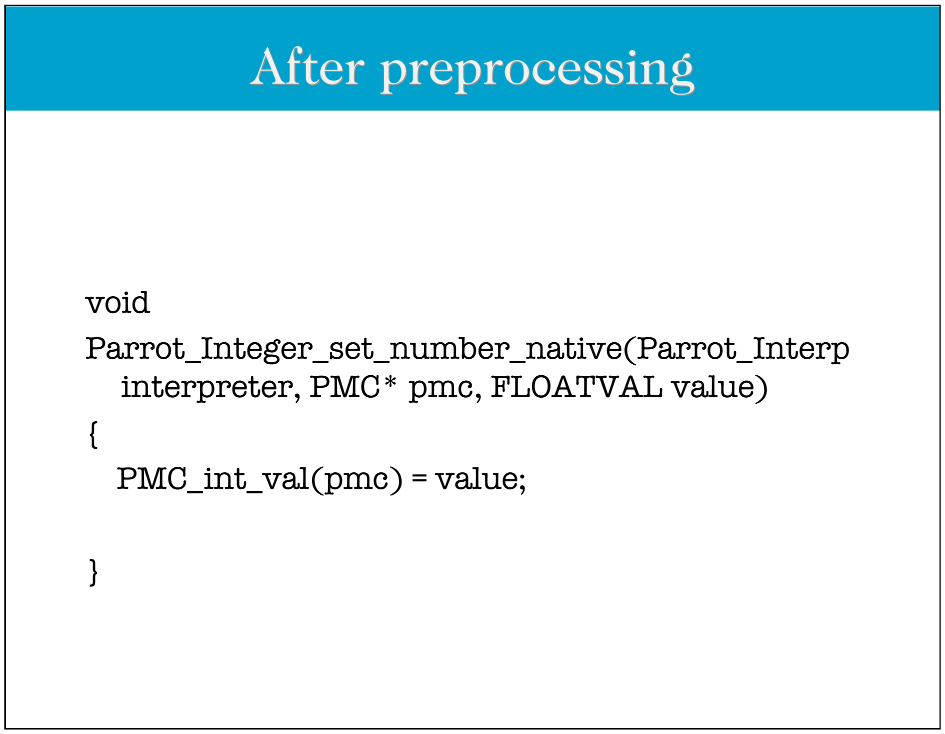
Before preprocessing

```
void set_number_native(FLOATVAL value) {
    PMC_int_val(SELF) = value;
  }
```

Simple, huh?

This is the function for an integer PMC type--you can tell since the macro, PMC_int_val, doesn't look like it's setting a number. What it is, in fact, is accessing the integer cache slot in the PMC. Each PMC has space to cache either an integer, float, or string, something we've found from perl 5 that makes a big performance win. (Granted, in perl 5 there's enough cache to store an int, float, *and* string, but we thought that would be a bit of overkill)

The point was to not have to bother with any crud that's required but uninteresting, The preprocessor loads that in for you, as you can see in the next slide.



After preprocessing

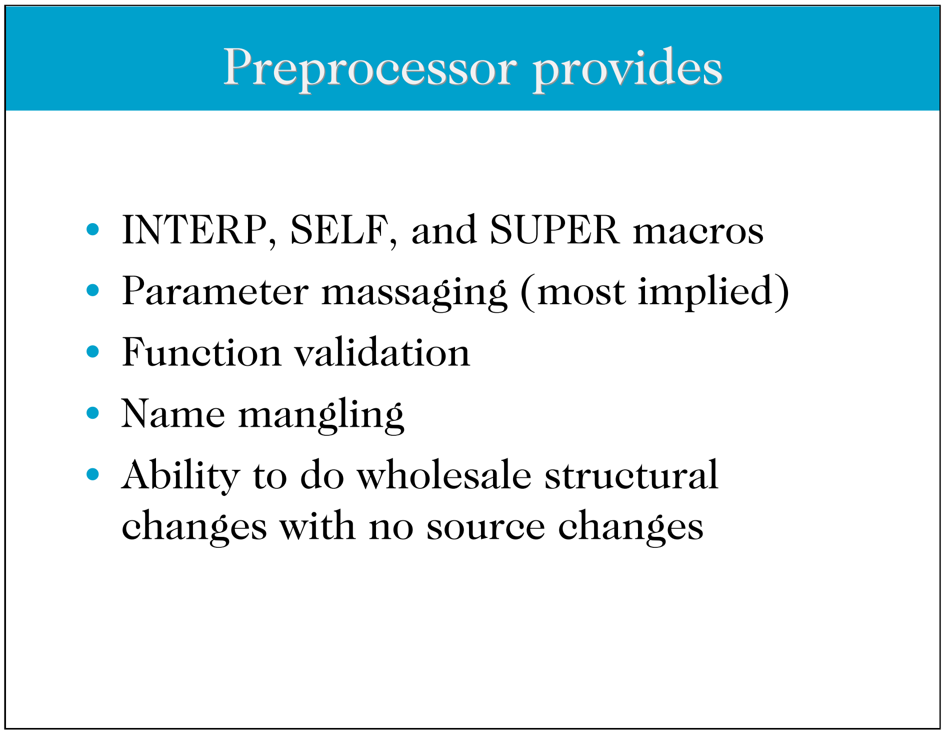
```
void
Parrot_Integer_set_number_native(Parrot_Interp
   interpreter, PMC* pmc, FLOATVAL value)
{
   PMC_int_val(pmc) = value;

}
```

Here you can see that first we've mangled the name. What was set_number_native is now Parrot_Integer_set_number_native. The Parrot_ part is in everything, the Integer_ part is the type name. We mangle like this because on some platforms all names are exposed and, well, we'd rather not collide. That's unpleasant.

The first two parameters aren't specified in the original form, as you can see back on the previous slide. Every single vtable function gets them, and in this order, so forcing them to be written is a waste. Bleah. That's what computers are for.

Anyway, there you go. Reasonably simple.



Preprocessor provides

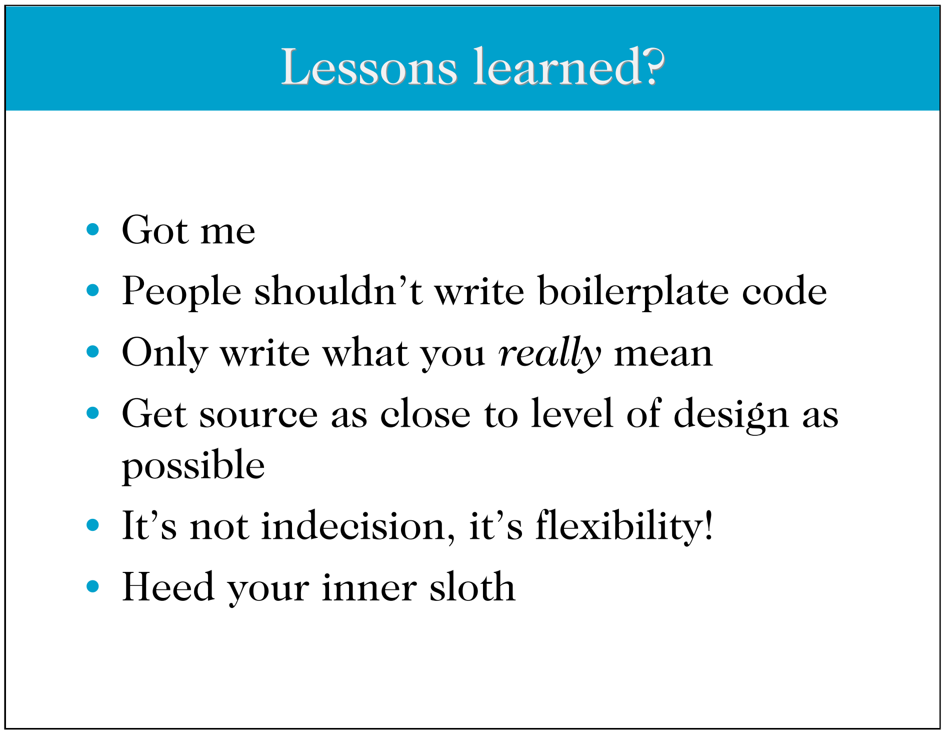
- INTERP, SELF, and SUPER macros
- Parameter massaging (most implied)
- Function validation
- Name mangling
- Ability to do wholesale structural changes with no source changes

The three macros INTERP, SELF, and SUPER aren't actually C macros, they're our custom preprocessor macros. Not that it makes a huge amount of difference, but we figure that since we're doing so much preprocessing we might as well do that as well. Besides, perl's a better language for this sort of thing than C's preprocessor is.

Most of the points are self-evident, but the last one bears some explanation, since there's some real-world use there.

Originally all the basic binary operations (add, subtract, multiply, divide, and so on) were in the vtable of the PMCs. When you did one of these things we called into the vtable of the left-hand PMC in the operation, and relied on it to Do The Right Thing. Well… turns out that most of the time the right thing to do is call into the MMD system. We were looking at 90%+ of the binary calls not actually doing anything, just immediately redispatching. If we yanked those functions out of the vtable and had the engine itself just automatically do MMD dispatching we'd do the same thing in the overwhelming majority of the cases, and skip the extra function call. So we did, which also shrank the vtable structure, for a memory win.

But… we had a fair amount of code for PMCs in the old style, with binary functions that actually did something besides redispatch, and it's nice to be able to stuff in default functions for binary operations into the PMC anyway, so… we taught the preprocessor how to handle these. From the programmer's standpoint nothing had changed--you still wrote the binary 'vtable' functions as normal. But now, rather than stuffing them into the vtable, the preprocessor extracts them and registers them as the default MMD function for the PMC. Woohoo!



Lessons learned?

- Got me
- People shouldn't write boilerplate code
- Only write what you *really* mean
- Get source as close to level of design as possible
- It's not indecision, it's flexibility!
- Heed your inner sloth

Yeah, we should've learned something--life's a long journey of discovery, blah, blah, blah. And I suppose we *did* learn something. Granted, what we learned is that we're lazy, bore easily, and like to fiddle, but know thyself, y'know? And, being the lazy, easily bored, fiddly folks that we are, we found it less work (and more interesting) to write the tools to do the gruntwork than it was to actually *do* the gruntwork.

Especially since it let us experiment with all sorts of things (new cores, new ways of generating opcode functions from source, ways to recover from bad design decisions without actually doing much work) reasonably easily. All we had to do was abuse the preprocessor code once, rather than having to go through and edit each and every opcode function, pmc source file, runloop, or whatever.

Questions?

?