This is the first part. It assumes familiarity with Plain English and the CAL-4700 development environment. Let me know how you do with it. Ask any questions that come to mind.

When developing a compiler and integrated development environment, we need to keep three "generations" in mind:

The CAL-4700: The program we're using to do the development.

The CAL-4700's son: The program we're developing.

The CAL-4700's son's daughter: The program we hope to compile with our new program.

So let's get to it.

1. Start up the CAL-4700.exe.

2. Find the CAL-4700 directory, and use the Duplicate command to make a copy. When asked to name the copy, enter "CAL-4700's son" just like that, but without the quotes. This will keep the two directories next to each other in the Finder, and will be a constant reminder of the son's roots.

3. Open up the new "CAL-4700's son" directory, and delete the CAL-4700.exe.

4. Open up the six source files in the son's directory. The quick way is to just drag-select them all, and hit ENTER.

5. In the Desktop file, find this line...

Put the lightest gray color into the desktop color.

...and change it to read:

Put the lightest sky color into the desktop color.

6. Now it's time to geld our boy, so he can grow a new pair, bit by bit. In the Compiler file, use the Select All command followed by the Comment command to "comment out" the whole file. All the text will turn a delightful shade of blue.

The CAL's son will no longer compile or run because certain types and routines that are defined in the compiler file are referenced elsewhere in the program. So we have to put those things back. And while we're doing that, we're going to put most of the compiler's "infrastructure" and "testing" routines back as well.

You can use the usual "find" procedure (CTRL-HOME, CTRL-F, start typing) to locate the stuff we need to put back, drag down the left edge of the work area to select the text, then use the Uncomment command (or CTRL-U) to do the deed. You should read over everything we put back, and ask questions about anything that you don't understand.

Now let's do it.

7. We'll be needing our error handling stuff (one type and three routines), specifically:

An abort message is a string.

To abort with a string:  
If the compiler's abort flag is set, exit.  
Set the compiler's abort flag.  
Put the string into the compiler's abort message.

To abort with a string and a byte pointer:  
If the compiler's abort flag is set, exit.  
Set the compiler's abort flag.  
Put the string into the compiler's abort message.  
Find the compiler's abort path and the compiler's abort row# given the byte pointer.  
If the compiler's abort path is blank, exit.  
Extract a file name from the compiler's abort path.  
Prepend "Error in " then the file name then ". " to the compiler's abort message.

To abort with a string and a rider:  
If the rider's token is not blank, abort with the string and the rider's token's first; exit.  
Abort with the string and the rider's source's last.

8. We'll also be needing all of our timers. There are 15 of them, corresponding to the 15 time-consuming steps in the compiler process. They look like this...

The adding built-in memory routines timer is a timer.

The adding built-in startup routine timer is a timer.

The addressing timer is a timer.

The calculating timer is a timer.

The compiling routine bodies timer is a timer.

The compiling routine headers timer is a timer.  
The indexing utilities timer is a timer.  
The linking timer is a timer.

The offsetting timer is a timer.

The loading timer is a timer.  
The resolving globals timer is a timer.

The resolving types timer is a timer.  
The scanning timer is a timer.

The transmogrifying timer is a timer.

The writing timer is a timer.

...and they're splattered all through the file, but you can easily put the first one back by typing CTRL-HOME, CTRL-F, "timer is a timer" (without the quotes), select and uncomment the line with END SHIFT-HOME CTRL-U. You can use CTRL-N followed by END SHIFT-HOME CTRL-U for the rest.

9. Find this type definition and put it back as well:

An address is a number.

10. These two routines below should also be put back. They're used to conveniently format the listing file.

To advance a buffer:  
Append the return byte to the buffer.  
Append the linefeed byte to the buffer.

To advance a buffer (twice):  
Advance the buffer.   
Advance the buffer.

11. The "append" routines below are also needed to format the listing file. There are of 10 them, but there are a handful of intervening "append" routines that are used for other purposes that we'll put back later.

To append an entry to a buffer (with separator):  
If the entry is nil, append "" to the buffer (with separator); exit.  
Append the entry's name to the buffer (with separator).

To append a flag to a buffer (with separator):  
Convert the flag to a string.  
Append the string to the buffer (with separator).

To append a number to a buffer (as hex with separator):  
Convert the number to a nibble string.  
Append the nibble string to the buffer (with separator).

To append a number to a buffer (with separator):  
Convert the number to a string.  
Append the string to the buffer (with separator).

To append a ratio to a buffer (with separator):  
Convert the ratio to a string.  
Append the string to the buffer (with separator).

To append a routine to a buffer (with separator):  
If the routine is nil, append "" to the buffer (with separator); exit.  
Append the routine's moniker to the buffer (with separator).

To append a string to a buffer (with separator):  
Append the string to the buffer.  
Append "/" to the buffer.

To append a tag to a buffer (as a fragment tag string with separator):  
If the tag is the break tag, append "break" to the buffer (with separator); exit.  
If the tag is the call external tag, append "call external" to the buffer (with separator); exit.  
If the tag is the call indirect tag, append "call indirect" to the buffer (with separator); exit.  
If the tag is the call internal tag, append "call internal" to the buffer (with separator); exit.  
If the tag is the dereference tag, append "dereference" to the buffer (with separator); exit.  
If the tag is the end if tag, append "end if" to the buffer (with separator); exit.  
If the tag is the epilog tag, append "epilog" to the buffer (with separator); exit.  
If the tag is the exit tag, append "exit" to the buffer (with separator); exit.  
If the tag is the finalize tag, append "finalize" to the buffer (with separator); exit.  
If the tag is the increment tag, append "increment" to the buffer (with separator); exit.  
If the tag is the intel tag, append "intel" to the buffer (with separator); exit.  
If the tag is the jump false tag, append "jump false" to the buffer (with separator); exit.  
If the tag is the load address tag, append "load address" to the buffer (with separator); exit.  
If the tag is the load eax tag, append "load eax" to the buffer (with separator); exit.  
If the tag is the loop tag, append "loop" to the buffer (with separator); exit.  
If the tag is the not tag, append "not" to the buffer (with separator); exit.  
If the tag is the prolog tag, append "prolog" to the buffer (with separator); exit.  
If the tag is the push address tag, append "push address" to the buffer (with separator); exit.  
If the tag is the push value tag, append "push value" to the buffer (with separator); exit.  
If the tag is the save eax tag, append "save eax" to the buffer (with separator); exit.  
If the tag is the repeat tag, append "repeat" to the buffer (with separator); exit.  
If the tag is the routine address tag, append "routine address" to the buffer (with separator); exit.  
Append "?" to the buffer (with separator).

To append a type to a buffer (with separator):  
If the type is nil, append "" to the buffer (with separator); exit.  
Append the type's name to the buffer (with separator).

To append a variable to a buffer (with separator):  
If the variable is nil, append "" to the buffer (with separator); exit.  
Append the variable's name to the buffer (with separator).

12. Woohoo! Done with the "A"s. Now we need to put our tags back, which are splattered around like the timers. There are 22 of them. Look for "is a tag equal to" and use the same END SHIFT-HOME CTRL-U keyboard sequence to make quick work of putting them back. Don't forget the tag type, shown here at the top:

A tag is a number.

The break tag is a tag equal to 1.

The call external tag is a tag equal to 2.

The call indirect tag is a tag equal to 3.

The call internal tag is a tag equal to 4.  
The dereference tag is a tag equal to 5.

The end if tag is a tag equal to 6.  
The epilog tag is a tag equal to 7.

The exit tag is a tag equal to 8.  
The finalize tag is a tag equal to 9.  
The increment tag is a tag equal to 10.  
The intel tag is a tag equal to 11.  
The jump false tag is a tag equal to 12.  
The load address tag is a tag equal to 13.

The load eax tag is a tag equal to 14.  
The loop tag is a tag equal to 15.  
The not tag is a tag equal to 16.  
The prolog tag is a tag equal to 17.  
The push address tag is a tag equal to 18.

The push value tag is a tag equal to 19.  
The repeat tag is a tag equal to 21.  
The routine address tag is a tag equal to 22.  
The save eax tag is a tag equal to 20.

While you're doing that, think about this: those 22 tags define the *only* things the compiler generates machine code to do. Which means the whole CAL (and every program written with the CAL) runs on just 22 machine-code "instructions." Wouldn't it be great if Intel built CPUs with this (and Aunt Tilly) in mind? Fewer transistors, lower costs, and no cooling fans necessary!

13. Done with the "B"s. And that brings us to the heart of the whole compiler:

To compile a directory:  
Abort with "I'm not old enough to make babies.".  
\Compile the directory (start).  
\Compile the directory (load the source files).  
\Compile the directory (scan the source files).  
\Compile the directory (resolve the types).  
\Compile the directory (resolve the globals).  
\Compile the directory (compile the headers of the routines).  
\Compile the directory (calculate lengths and offsets of types).  
\Compile the directory (add the built-in memory routines).  
\Compile the directory (index the routines for utility use).  
\Compile the directory (compile the bodies of the routines).  
\Compile the directory (add and compile the built-in startup routine).  
\Compile the directory (offset parameters and variables).  
\Compile the directory (address).  
\Compile the directory (transmogrify).  
\Compile the directory (link).  
\Compile the directory (write the exe).  
\Compile the directory (stop).

In this case, don't uncomment the whole routine. Uncomment only the header, and add the "abort" line you see above. Note the two periods in the abort message, one inside and one outside the string. Leave the rest blue.

Isn't it handy to be able to see, at a glance, which parts of the program are commented, and which parts are not? Now *that's* a proper use of color-coding.

14. Time to put back the compiler record:

The compiler has  
a directory,  
a timer,  
an abort flag,  
an abort message,  
an abort path,  
an abort row#,  
a name count,  
an exe buffer,  
a listing buffer.

15. Then a couple of trivial "deciders":

To decide if a byte is any mark:  
If the byte is the period byte, say yes.  
If the byte is the comma byte, say yes.  
If the byte is the semi-colon byte, say yes.  
If the byte is the colon byte, say yes.  
If the byte is the exclamation  byte, say yes.  
If the byte is the question-mark byte, say yes.  
Say no.

To decide if a string is any flag literal:  
If the string is "yes", say yes.  
If the string is "no", say yes.  
Say no.

16. Then the "entry" type:

An entry is a thing with  
A name address,  
A thunk address,  
A name,  
An address.

There's one "entry" for each DLL routine that we call.

17. Now we're up to the "F"s. Let's put back the finalize routine:

To finalize the compiler:  
Destroy the utility index.  
Destroy the routine index.  
Destroy the literal index.  
Destroy the global index.  
Destroy the type index.  
Destroy the imports.  
Destroy the routines.  
Destroy the literals.  
Destroy the globals.  
Destroy the types.  
Destroy the source files.

18. And now, the "fragment" type, that makes use of all those tags we put back earlier:

A fragment is a thing with  
A tag [break, call external, call indirect, call internal, dereference, end if, epilog, exit, finalize, increment, intel, jump false, load address, load eax, loop, not, prolog, push address, push value, save eax, repeat, routine address],  
A variable (reference) [call indirect, dereference, increment, load address, load eax, pop, push address, push value, routine address, save eax],  
Another variable (reference) [load address],  
A routine (reference) [call internal, routine address],  
An entry (reference) [call external],  
A number [increment],  
An address,  
A hex string called code.

19. Next, the "global body" type. This is a substring that spans the part of a global variable definition in the source code that we're not ready to process until later:

A global body is a substring.

20. Time for more keyboarding fun. All our indexes need to be put back. There are only 5.

The global index is an index.

The literal index is an index.

The routine index is an index.  
The type index is an index.  
The utility index is an index.

While you're putting those back, think about this: All a compiler really has to do, as it wades through source code, is to collect the information it needs to put GLOBALS, LITERALS, and ROUTINES in the right places in an EXE file. It uses TYPES to tell it how big things are, but types don't end up anywhere in the EXE. The rest is decoration. If you're wondering about the Utility Index, it's used to allow generic routines to operate on types that aren't formally specified in their headers. For example, the "number" type and the "pointer" type are distinct in our Noodle; but pointers are really just numbers, so routines like "To add a number to another number" ought to work on pointers as well as numbers. So, as you'll see later, we do a little type trickery, using the Utility Index, to allow that to happen.

21. Now put back these three guys:

A global is a variable.

A field is a variable.

The globals are some globals.

The first two are types that say global variables and fields are a lot like any other variables. The third is a global variable itself: it's the list on which we hang all the global variables we find.

22. Moving along. Put these three things back:

An import header is a record with  
An address called original first thunk,  
A number called timestamp,  
An address called forwarder chain,  
An address called name memory address,  
An address called first thunk memory address.

An import is a thing with  
A name address,  
A header address,  
A name,  
An import header,  
Some entries.

The imports are some imports.

An "import" is a DLL. That's what Windows calls them. Ugly stuff.

23. Let's not forget this silly little guy...

To initialize the compiler:

...who does nothing at all. We put it in, and left it in, just to make Desktop code up above symmetrical.

24. And now we put back the code we need for testing. Everything that starts with "To List..." except the file lines that list indexes (see below). Leave them out; we haven't created the indices yet, so they won't work and the thing will blow away. The listing stuff is all in a bunch; just find the first routine and drag down until you've got them all. But make sure you re-comment those five lines.

To list:  
Clear the compiler's listing.  
List the compiler's abort message in the compiler's listing.  
List the types under "TYPES:" in the compiler's listing.  
List the globals under "GLOBALS:" in the compiler's listing.  
List the literals under "LITERALS:" in the compiler's listing.  
List the routines under "ROUTINES:" in the compiler's listing.  
\List the type index under "TYPE INDEX:" in the compiler's listing.  
\List the global index under "GLOBAL INDEX:" in the compiler's listing.  
\List the literal index under "LITERAL INDEX:" in the compiler's listing.  
\List the routine index under "ROUTINE INDEX:" in the compiler's listing.  
\List the utility index under "UTILITY INDEX:" in the compiler's listing.  
List the imports under "IMPORTS:" in the compiler's listing.  
List the source files under "SOURCE FILES:" in the compiler's listing.  
List the timers under "TIMERS:" in the compiler's listing.  
Write the compiler's listing to the compiler's listing path.

To list an abort message in a buffer:  
If the abort message is blank, exit.  
Append "COMPILE ABORTED - LISTING INCOMPLETE" to the buffer.  
Advance the buffer (twice).  
Append the abort message to the buffer.  
Advance the buffer (twice).

To list a bucket in a buffer:  
If the bucket's refers are empty, exit.  
Append "/bucket" to the buffer (with separator).  
Advance the buffer.  
List the bucket's refers in the buffer.  
Advance the buffer.

To list some entries in a buffer:  
Get an entry from the entries.  
If the entry is nil, exit.  
List the entry in the buffer.  
Repeat.

To list an entry in a buffer:  
If the entry is nil, exit.  
Append "/entry" to the buffer (with separator).  
Append the entry's name to the buffer (with separator).  
Append the entry's address to the buffer (as hex with separator).  
Advance the buffer.

To list a fragment in a buffer:  
Append "/fragment" to the buffer (with separator).  
Append the fragment's tag to the buffer (as a fragment tag string with separator).  
Append the fragment's variable to the buffer (with separator).  
Append the fragment's other variable to the buffer (with separator).  
Append the fragment's routine to the buffer (with separator).  
Append the fragment's entry to the buffer (with separator).  
Append the fragment's number to the buffer (as hex with separator).  
Append the fragment's address to the buffer (as hex with separator).  
Convert the fragment's code to a nibble string.  
Append the nibble string to the buffer (with separator).  
Advance the buffer.

To list some fragments in a buffer:  
Get a fragment from the fragments.  
If the fragment is nil, exit.  
List the fragment in the buffer.  
Repeat.

To list an import in a buffer:  
If the import is nil, exit.  
Append "/import" to the buffer (with separator).  
Append the import's name to the buffer (with separator).  
Advance the buffer.  
List the import's entries in the buffer.  
Advance the buffer.

To list some imports in a buffer:  
Get an import from the imports.  
If the import is nil, exit.  
List the import in the buffer.  
Repeat.

To list some imports under a string in a buffer:  
Append the string to the buffer.  
Advance the buffer (twice).  
If the imports are empty, exit.  
List the imports in the buffer.

To list an index in a buffer:  
Append "/" then the index's used bucket count then " buckets" to the buffer (with separator).  
Append the index's count then " refers" to the buffer (with separator).  
Advance the buffer (twice).  
Loop.  
Get a bucket given the index.  
If the bucket is nil, exit.  
List the bucket in the buffer.  
Repeat.

To list an index under a string in a buffer:  
Append the string to the buffer.  
Advance the buffer (twice).  
If the index is empty, exit.  
List the index in the buffer.

To list a refer in a buffer:  
If the refer is nil, exit.  
Append "/refer" to the buffer (with separator).  
Append the refer's string to the buffer.  
If the refer's pointer is nil, append "..." to the buffer.  
Append "/" to the buffer.  
Advance the buffer.

To list some refers in a buffer:  
Get a refer from the refers.  
If the refer is nil, exit.  
List the refer in the buffer.  
Repeat.

To list a routine in a buffer:  
If the routine is nil, exit.  
Append "/routine" to the buffer (with separator).  
Append the routine's moniker to the buffer (with separator).  
Append the routine's compiled flag to the buffer (with separator).  
Append the routine's callback flag to the buffer (with separator).  
Append the routine's decider flag to the buffer (with separator).  
Append the routine's function flag to the buffer (with separator).  
Append the routine's employs moniker to the buffer (with separator).  
Append the routine's parameter size to the buffer (with separator).  
Append the routine's local size to the buffer (with separator).  
Append the routine's address to the buffer (as hex with separator).  
Advance the buffer.  
List the routine's parameters in the buffer.  
List the routine's locals in the buffer.  
List the routine's fragments in the buffer.  
Advance the buffer.

To list some routines in a buffer:  
Get a routine from the routines.  
If the routine is nil, exit.  
List the routine in the buffer.  
Repeat.

To list some routines under a string in a buffer:  
Append the string to the buffer.  
Advance the buffer (twice).  
If the routines are empty, exit.  
List the routines in the buffer.

To list a source file in a buffer:  
If the source file is nil, exit.  
Append "/source file" to the buffer (with separator).  
Append the source file's path to the buffer (with separator).  
Advance the buffer.

To list some source files in a buffer:  
Get a source file from the source files.  
If the source file is nil, exit.  
List the source file in the buffer.  
Repeat.

To list some source files under a string in a buffer:  
Append the string to the buffer.  
Advance the buffer (twice).  
If the source files are empty, exit.  
List the source files in the buffer.  
Advance the buffer.

To list a timer given a string in a buffer:  
Append "/timer" to the buffer (with separator).  
Append the string to the buffer (with separator).  
Append the timer's string to the buffer (with separator).  
Advance the buffer.

To list the timers under a string in a buffer:  
Append the string to the buffer.  
Advance the buffer (twice).  
List the loading timer given "loading" in the buffer.  
List the scanning timer given "scanning" in the buffer.  
List the resolving types timer given "resolving types" in the buffer.  
List the resolving globals timer given "resolving globals" in the buffer.  
List the compiling routine headers timer given "compiling routine headers" in the buffer.  
List the calculating timer given "calculating" in the buffer.  
List the adding built-in memory routines timer given "adding built-in memory routines" in the buffer.  
List the indexing utilities timer given "indexing utilities" in the buffer.  
List the compiling routine bodies timer given "compiling routine bodies" in the buffer.  
List the adding built-in startup routine timer given "adding built-in startup routine" in the buffer.  
List the offsetting timer given "offsetting" in the buffer.  
List the addressing timer given "addressing" in the buffer.  
List the transmogrifying timer given "transmogrifying" in the buffer.  
List the linking timer given "linking" in the buffer.  
List the writing timer given "writing" in the buffer.  
List the compiler's timer given "total" in the buffer.  
Advance the buffer.

To list a type in a buffer:  
If the type is nil, exit.  
Append "/type" to the buffer (with separator).  
Append the type's name to the buffer (with separator).  
Append the type's plural name to the buffer (with separator).  
Append the type's length to the buffer (as hex with separator).  
Append the type's base name to the buffer (with separator).  
Append the type's base type to the buffer (with separator).  
Append the type's target name to the buffer (with separator).  
Append the type's target type to the buffer (with separator).  
Append the type's scale to the buffer (with separator).  
Advance the buffer.  
List the type's fields in the buffer.  
Advance the buffer.

To list some types in a buffer:  
Get a type from the types.  
If the type is nil, exit.  
List the type in the buffer.  
Repeat.

To list some types under a string in a buffer:  
Append the string to the buffer.  
Advance the buffer (twice).  
If the types are empty, exit.  
List the types in the buffer.

To list a variable in a buffer:  
If the variable is nil, exit.  
Append "/variable" to the buffer (with separator).  
Append the variable's kind to the buffer (with separator).  
Append the variable's compiled flag to the buffer (with separator).  
Append the variable's name to the buffer (with separator).  
Append the variable's nickname to the buffer (with separator).  
Append the variable's type name to the buffer (with separator).  
Append the variable's type to the buffer (with separator).  
Append the variable's address [or offset] to the buffer (as hex with separator).  
Append the variable's by-value flag to the buffer (with separator).  
Append the variable's count to the buffer (with separator).  
Append the variable's reference flag to the buffer (with separator).  
Append the variable's redefinition target name to the buffer (with separator).  
Append the variable's literal to the buffer (with separator).  
Convert the variable's data to a nibble string.  
Append the nibble string to the buffer (with separator).  
Advance the buffer.

To list some variables in a buffer:  
Get a variable from the variables.  
If the variable is nil, exit.  
List the variable in the buffer.  
Repeat.

To list some variables under a string in a buffer:  
Append the string to the buffer.  
Advance the buffer (twice).  
If the variables are empty, exit.  
List the variables in the buffer.  
Advance the buffer.

25. Another type and an associated global list (we treat literals like global variables):

A literal is a variable.

The literals are some literals.

26. A type for local variables:

A local is a variable.

There's no global list of local variables because they hang on separate lists inside each routine record.

27. And here's a goodie:

A locus is a byte pointer.

A "locus" points to the very spot in the source code where an error is discovered. That, plus a fully qualified path, lets the Desktop find and position the programmer on the offending line of code when an error is discovered.

28. Routines are internally known by their "monikers" which are made up of monikettes, where each monikette can be a word, a bunch of words, a type, or a variable. So we need this stuff put back:

A moniker is a string.

A monikette is a thing with  
a string,  
a type (reference),  
a variable (reference),  
\ for bubbling  
a current type (reference),  
a current substring.

29. Along with this type definition:

A nickname is a name.

A "nickname" is the short name for a variable. "the left", for example, instead of "the left coordinate". It's typically the adjective that precedes the type name noun.

30. Now one more kind of variable:

A parameter is a variable.

Again, there's no global list of parameters because they're kept on separate lists in each routine record.

31. And that brings us to a couple of "function-style routines" (routines that return a value and that can thus be used as expressions):

To put the compiler's exe path into a path:  
Extract a designator from the compiler's directory.  
Remove any trailing backslash from the designator.  
Put the compiler's directory then the designator then ".exe" into the path.

To put the compiler's listing path into a path:  
Extract a designator from the compiler's directory.  
Remove any trailing backslash from the designator.  
Put the compiler's directory then the designator then ".lst" into the path.

These routines let us say "the compiler's exe path" and "the compiler's listing path" as if those two things were fields in the compiler record. Kind of like "derived data" or a "calculated field" in a database.

32. Now some routine stuff:

A routine body is a substring.

A routine header is a substring.

The routine index is an index.

A routine is a thing with  
A locus (reference),  
A callback flag,  
A decider flag,  
A function flag,  
A compiled flag,  
A nickname index,  
A moniker, some monikettes, some parameters, a parameter size,  
An employs moniker,  
Some locals, a local size,  
A routine header, a header string,  
A routine body, a body string,  
Some fragments,  
An address.

A routine reference is some monikettes.

The routines are some routines.

33. Almost there. Up to the "S"s. Let's not forget the source code! Put these back in:

A source file is a thing with a path and a buffer.

The source files are some source files.

34. We need our type stuff, too:

A type is a thing with  
A locus (reference),  
A name, a plural name,  
A partial moniker,  
A length,  
A base name, a base type (reference),  
A target name, a target type (reference) [pointers only],  
A scale ratio,  
Some fields [records only],  
A cooking flag,  
A optional info resolved flag.

A type name is a name.

The types are some types.

35. And last but not least, our multi-purpose variable record:

A variable is a thing with  
A locus (reference),  
A kind [literal, global, local, parameter, scratch],  
A compiled flag,  
A name, a nickname, a type name,  
A type (reference), \ actual type on literal, global, local; dereferenced type on parameters and scratches  
An address [globals and literals only] or   
An offset at the address [locals, parameters, and fields only],  
A count [fields only],  
A reference flag [fields only],  
A redefinition target name [fields only],  
A by-value flag [parameters only],  
An global body [globals only],  
An initializer string [globals only],  
A literal (reference) [globals only - constant with which to initalize the global],  
A hex string called data [literals only].

And that's *almost* it for this time around. Let's test.

36. Run the CAL-4700's son. Assuming it worked, congratulations are in order. It's a boy! And the blue desktop will serve to remind us which generation we're in. ALT-TAB back and forth a couple of times.

37. Now let's try to make a grand-baby. Working in the CAL-4700's son (blue desktop), find the CAL-4700's son directory and use the Duplicate command to make a copy. Call it "CAL-4700's son's daughter" too keep the family together in the Finder.

38. Now, still in the blue, open up the daughter and delete the CAL-4700's son.exe.

39. Still in the blue, open up the six source files in the daughter directory.

40. Change the line in the daughter's desktop that says...

Put the lightest sky color into the desktop color.

...to say...

Put the lightest red color into the desktop color.

41. Now, still in the blue, try to compile or run or list the daughter. You should get this message:

"I'm not old enough to make babies."

Ta Da! End of Part 01.

Here are the rest of the parts. A graphic or two may have gotten inadvertently deleted when I combined them; let me know if it looks like something important is missing.

So where were we? Ah, yes.

42. Close up the Son of CAL (if you haven't already) and get back to the tasteful and non-distracting gray of the CAL-4700. Then find the CAL-4700's son directory and open up the son's six source files.

43. Now find the "compile a directory" routine, move our temporary "abort" line down a notch, and uncomment the "Compile a directory (start)" step, like so:

To compile a directory:  
Compile the directory (start).  
Abort with "I'm not old enough to make babies.".  
\Compile the directory (load the source files).  
\Compile the directory (scan the source files).  
\Compile the directory (resolve the types).  
\Compile the directory (resolve the globals).  
\Compile the directory (compile the headers of the routines).  
\Compile the directory (calculate lengths and offsets of types).  
\Compile the directory (add the built-in memory routines).  
\Compile the directory (index the routines for utility use).  
\Compile the directory (compile the bodies of the routines).  
\Compile the directory (add and compile the built-in startup routine).  
\Compile the directory (offset parameters and variables).  
\Compile the directory (address).  
\Compile the directory (transmogrify).  
\Compile the directory (link).  
\Compile the directory (write the exe).  
\Compile the directory (stop).

44. And put these guys back in:

To compile a directory (start):  
Show status "Starting...".  
Finalize the compiler.  
Start the compiler's timer.  
Initialize the compiler given the directory.  
Add the built-in types.

A built-in type is a type.

We "Finalize the compiler" to clean up anything that might be hanging around from a previous compile.

45. As you can see, we're going to need this routine, too:

To initialize the compiler given a directory:  
Put the directory into the compiler's directory.  
Put 0 into the compiler's name count.  
Clear the compiler's abort flag.  
Clear the compiler's abort message.  
Clear the compiler's abort path.  
Put 0 into the compiler's abort row#.  
Void the current routine.  
Create the type index with 4027 [buckets].  
Create the global index with 4027 [buckets].  
Create the literal index with 4027 [buckets].  
Create the routine index with 7919 [buckets].  
Create the utility index with 4027 [buckets].

46. Now that we've got our indices created, we can uncomment those 5 blue lines in the List routine, as well:

To list:  
Clear the compiler's listing.  
List the compiler's abort message in the compiler's listing.  
List the types under "TYPES:" in the compiler's listing.  
List the globals under "GLOBALS:" in the compiler's listing.  
List the literals under "LITERALS:" in the compiler's listing.  
List the routines under "ROUTINES:" in the compiler's listing.  
List the type index under "TYPE INDEX:" in the compiler's listing.  
List the global index under "GLOBAL INDEX:" in the compiler's listing.  
List the literal index under "LITERAL INDEX:" in the compiler's listing.  
List the routine index under "ROUTINE INDEX:" in the compiler's listing.  
List the utility index under "UTILITY INDEX:" in the compiler's listing.  
List the imports under "IMPORTS:" in the compiler's listing.  
List the source files under "SOURCE FILES:" in the compiler's listing.  
List the timers under "TIMERS:" in the compiler's listing.  
Write the compiler's listing to the compiler's listing path.

47. We're going to be needing this global variable:

The current routine is a routine.

48. And these two routines as well:

To add the built-in types:  
Add a built-in type given "byte" and "bytes" and 1.  
Add another built-in type given "record" and "records" and 0.

To add a built-in type given a name and a plural name and a length:  
Add the built-in type to the types given the name and the plural name and the name.  
Put the length into the built-in type's length.  
Put the built-in type into the built-in type's base type.  
Index the built-in type.

Note that *byte* and *record* are the only types defined in the compiler itself; all the other "built-in" types (like *number* and *string* and *ratio*, etc) are in the Noodle.

49. We add types in a number of places when we're compiling, and sometimes it's convenient to do it one way, sometimes another. So let's put back these 3 "add a type" helper routines as well:

To add a type to some types given a locus:  
Create the type.  
Append the type to the types.  
Put the locus into the type's locus.

To add a type to some types given a name and a plural name and a base name:  
Add the type to the types given the name and the plural name and the base name and nil.

To add a type to some types given a name and a plural name and a base name and a locus:  
Create the type.  
Append the type to the types.  
Put the locus into the type's locus.  
Put the name into the type's name.  
Put the plural name into the type's plural name.  
Put the base name into the type's base name.

50. We'll also need the routine below, that allocates and initializes a type record in the Heap:

To create a type:  
Allocate memory for the type.  
Put 0/1 into the type's scale.  
Put -1 into the type's length.

51. And we'll need these two routines to index our types:

To index a type:  
If the type is nil, exit.  
If the compiler's abort flag is set, exit.  
Index the type given the type's name.

To index a type given a name:  
If the type is nil, exit.  
If the compiler's abort flag is set, exit.  
Find an existing type given the name and the type index.  
If the existing type is nil, index the type given the name and the type index; exit.  
Put the type's locus into a locus.  
If the locus is nil, put the existing type's locus into the locus. \ point the error to a type in the source code, not a generated type  
Abort with "I think I've seen this '" then the name then "' type somewhere before; it must be a duplicate." and the locus.

52. This is a little premature, but the abort in the above routine calls another abort routine that needs a little help from the three guys below, so we need to put them back, too:

To find a path and a row# given a byte pointer:  
Clear the path.  
Put 0 into the row#.  
Find a source file given the byte pointer.  
If the source file is nil, exit.  
Put the source file's path into the path.  
Find the row# given the source file and the byte pointer.

To find a source file given a byte pointer:  
If the byte pointer is nil, void the source file; exit.  
Loop.  
Get the source file from the source files.  
If the source file is nil, exit.  
If the byte pointer is less than the source file's buffer's first, repeat.  
If the byte pointer is greater than the source file's buffer's last, repeat.

To find a row# given a source file and a byte pointer:  
If the byte pointer is nil, put 0 into the row#; exit.  
Slap a substring on the source file's buffer.  
Put 1 into the row#.  
Loop.  
If the substring is blank, exit.  
If the substring's first is the byte pointer, exit.  
If the substring's first's target is the return byte, add 1 to the row#.  
Add 1 to the substring's first.  
Repeat.

53. And that's about it for this step. Let's run it as see what we've got. Press CTRL-R.

54. Now we're in the blue. Find and open the daughter's six files, then execute the List command (CTRL-L). The compile will still abort with our "I'm not old enough to make babies" message, but this time a listing file will be created.

55. On the far right tab, open up the daughter's directory, find the file with the ".lst" extension, and double-click it. You should see something like this at the top...

COMPILE ABORTED - LISTING INCOMPLETE

I'm not old enough to make babies.

TYPES:

/type/byte/bytes/00000001/byte/byte///0/

/type/record/records/00000000/record/record///0/

GLOBALS:

LITERALS:

ROUTINES:

TYPE INDEX:

/2 buckets/2 refers/

...which shows that our two built-in types got created and hung on the list properly, and that our type index is working, with 2 buckets used to hold 2 refers (one for each type).

And after a bunch of other stuff that we don't have, at the bottom of the listing, you'll find our timers...

TIMERS:

/timer/loading/0/  
/timer/scanning/0/  
/timer/resolving types/0/  
/timer/resolving globals/0/  
/timer/compiling routine headers/0/  
/timer/calculating/0/  
/timer/adding built-in memory routines/0/  
/timer/indexing utilities/0/  
/timer/compiling routine bodies/0/  
/timer/adding built-in startup routine/0/  
/timer/offsetting/0/  
/timer/addressing/0/  
/timer/transmogrifying/0/  
/timer/linking/0/  
/timer/writing/0/  
/timer/total/0/

...which tell us we don't have any performance bottlenecks yet. Sweet.

Next time, we'll find and load up the source files... Stay tuned.

---

Alrighty, then. Time to load up our source files.

56. Close up the Son of CAL (if you haven't already) and get back to the CAL-4700. Then find the CAL-4700's son directory and open up the son's six source files (if they're not already open).

57. Now find the "compile a directory" routine, move our temporary "abort" line down another notch, and uncomment the "Compile a directory (load the source files)" step, like so:

To compile a directory:  
Compile the directory (start).  
Compile the directory (load the source files).  
Abort with "I'm not old enough to make babies.".  
\Compile the directory (scan the source files).  
\Compile the directory (resolve the types).  
\Compile the directory (resolve the globals).  
\Compile the directory (compile the headers of the routines).  
\Compile the directory (calculate lengths and offsets of types).  
\Compile the directory (add the built-in memory routines).  
\Compile the directory (index the routines for utility use).  
\Compile the directory (compile the bodies of the routines).  
\Compile the directory (add and compile the built-in startup routine).  
\Compile the directory (offset parameters and variables).  
\Compile the directory (address).  
\Compile the directory (transmogrify).  
\Compile the directory (link).  
\Compile the directory (write the exe).  
\Compile the directory (stop).

58. Then put these 3 routines back in:

To compile a directory (load the source files):  
If the compiler's abort flag is set, exit.  
Show status "Loading...".  
Start the loading timer.  
Load the source files.  
Stop the loading timer.

To load a source file:  
If the source file is nil, exit.  
Load the source file's path into the source file's buffer.  
If the i/o error is not blank, abort with "Could not load the file named '" then the source file's path then "'."; exit.

To load some source files:  
Loop.  
If the compiler's abort flag is set, exit.  
Get an item from the compiler's directory.  
If the item is not found, break.  
If the item's kind is not "file", repeat.  
If the item's extension is not blank, repeat.  
If the item looks reamish, repeat.  
Add a source file to the source files given the item's path.  
Load the source file.  
Repeat.

As you know, the CAL expects a program's source code to be stored as one or more text files in a single directory. One of those files is usually a copy of the Noodle. It doesn't matter what order the files are in, and it doesn't matter what their names are, except that he will only attempt to compile files with no extension.

A diligent Plain English programmer, however, will typically use the CAL's Writer to author a user manual and other documentation for a program. Since it's convenient to keep such documentation in the same directory as the source code, and since such documentation files will not have an extension, we need the CAL to skip them when he's collecting source code to compile.

59. Internally, documents created with the CAL's writer are called "reams" (as in a "ream of paper"). And that's why, in the above routine, files that look "reamish" are passed over. Uncomment the decider routine shown below so the compiler will be able to recognize "reamish" files:

To decide if an item looks reamish:  
Load the item's path into a buffer.  
If the buffer starts with "ream cal", say yes.  
Say no.

60. We're also going to need these two routines:

To add a source file to some source files given a path:  
Create the source file.  
Append the source file to the source files.  
Put the path into the source file's path.

To create a source file:  
Allocate memory for the source file.

61. And we're done with this step. Let's see what we've got. Run the Son of CAL and when you've got the blue desktop up, find the son's daughter and open up her source files. Then create an updated listing file with CTRL-L.

62. Still in the blue, find and open the daughter's listing on the far right tab. The top part of the listing will be the same as before, but at the bottom you'll see something like this:

SOURCE FILES:

/source file/c:\cal-4700's son's daughter\the compiler/  
/source file/c:\cal-4700's son's daughter\the desktop/  
/source file/c:\cal-4700's son's daughter\the editor/  
/source file/c:\cal-4700's son's daughter\the finder/  
/source file/c:\cal-4700's son's daughter\the noodle/  
/source file/c:\cal-4700's son's daughter\the writer/

TIMERS:

/timer/loading/140/  
/timer/scanning/0/  
/timer/resolving types/0/  
/timer/resolving globals/0/  
/timer/compiling routine headers/0/  
/timer/calculating/0/  
/timer/adding built-in memory routines/0/  
/timer/indexing utilities/0/  
/timer/compiling routine bodies/0/  
/timer/adding built-in startup routine/0/  
/timer/offsetting/0/  
/timer/addressing/0/  
/timer/transmogrifying/0/  
/timer/linking/0/  
/timer/writing/0/  
/timer/total/140/

TA DA! Now we're (finally) ready to teach our boy how to read.

---

Now that we've got all our source files in memory, it's time to see what they have to say.

63. Close up the Son of CAL (if you haven't already) and get back to the CAL-4700. Then find the CAL-4700's son directory and open up the son's six source files (if they're not already open).

64. Now find the "compile a directory" routine, move our temporary "abort" line down another notch, and uncomment the "Compile a directory (scan the source files)" step, like so:

To compile a directory:  
Compile the directory (start).  
Compile the directory (load the source files).  
Compile the directory (scan the source files).  
Abort with "I'm not old enough to make babies.".  
\Compile the directory (resolve the types).  
\Compile the directory (resolve the globals).  
\Compile the directory (compile the headers of the routines).  
\Compile the directory (calculate lengths and offsets of types).  
\Compile the directory (add the built-in memory routines).  
\Compile the directory (index the routines for utility use).  
\Compile the directory (compile the bodies of the routines).  
\Compile the directory (add and compile the built-in startup routine).  
\Compile the directory (offset parameters and variables).  
\Compile the directory (address).  
\Compile the directory (transmogrify).  
\Compile the directory (link).  
\Compile the directory (write the exe).  
\Compile the directory (stop).

65. We'll be needing this guy to get us going:

To compile a directory (scan the source files):  
If the compiler's abort flag is set, exit.  
Show status "Scanning...".  
Start the scanning timer.  
Scan the source files.  
Stop the scanning timer.

And this little loop to make sure we scan all the source files in the project:

To scan some source files:  
If the compiler's abort flag is set, exit.  
Get a source file from the source files.  
If the source file is nil, exit.  
Scan the source file.  
Repeat.

66. A Plain English program is a collection of TYPES, GLOBALS, and ROUTINES, in any order. This is the top-level scanner that goes looking for those things:

To scan a source file:  
If the source file is nil, exit.  
If the compiler's abort flag is set, exit.  
Slap a rider on the source file's buffer.  
Move the rider (compiler rules).  
Loop.  
If the compiler's abort flag is set, exit.  
If the rider's token is blank, exit.  
If the rider's token is the start of any type, scan a type given the rider; repeat.  
If the rider's token is the start of any global, scan a global given the rider; repeat.  
If the rider's token is the start of any routine, scan a routine given the rider; repeat.  
Abort with "I was hoping for a definition but all I found was '" then the rider's token then "'." and the rider.  
Repeat.

67. The routine above needs a lot of help, first, to move that rider around, and secondly, to do some detailed scanning once he's found a type, a global, or a routine to scan. Here's one of the "move a rider" routines that he needs:

To move a rider (code rules - noise):  
Bump the rider.  
If the rider's source is blank, exit.  
If the rider's source's first's target is noise, repeat.

That guy simply skips any "noise" in the source file. When he's done, what's left of the rider's source will start with the next non-noise byte in the file, and the noise that was sucked up will be spanned by the rider's token. Noise is defined in the Noodle (not the compiler) like this:

To decide if a byte is noise:  
If the byte is less than or equal to the space byte, say yes.  
If the byte is the delete byte, say yes.  
If the byte is the non-breaking-space byte, say yes.  
If the byte is 129, say yes.  
If the byte is 141, say  yes.  
If the byte is 143, say yes.  
If the byte is 144, say yes.  
If the byte is 157, say yes.  
Say no.

68. Here's another "move a rider" routine we'll be needing:

To move a rider (code rules - comment):  
Bump the rider.  
If the rider's source is blank, exit.  
If the rider's source's first's target is not the return byte, repeat.

This guy assumes we're sitting on the start of a comment (the backslash character) and simply sucks up bytes until he hits a return byte (the end of a line). When he's done, what's left of the rider's source will start with that return byte, and the rider's token will span the whole comment, including the backslash.

69. We need a similar "move a rider" routine for remarks:

To move a rider (code rules - remark):  
If the rider's source is blank, exit.  
If the rider's source's first's target is the return byte, break.  
If the rider's source's first's target is the left-bracket byte, add 1 to a count.  
If the rider's source's first's target is the right-bracket byte, subtract 1 from the count.  
Bump the rider.  
If the count is 0, break.  
Repeat.

This time we're starting on the start of a remark (a left-bracket "["), and we skip everything up to a return byte (the end of a line) or a closing right-bracket ("]"), allowing for the possibility that the programmer may have remarks inside his remarks. When this routine is done, what's left of the rider's source will start with either a return byte, or the first character after the closing right-bracket, and the rider's token will span the entire remark.

70. We need a similar "move a rider" routine for qualifiers, which can also be "nested":

To move a rider (code rules - qualifier):  
If the rider's source is blank, exit.  
If the rider's source's first's target is the return byte, break.  
If the rider's source's first's target is the left-paren byte, add 1 to a count.  
If the rider's source's first's target is the right-paren byte, subtract 1 from the count.  
Bump the rider.  
If the count is 0, break.  
Repeat.

71. And another for strings, that may include doubled-up double-quotes to indicate imbedded double-quote marks:

To move a rider (code rules - string):  
Bump the rider.  
If the rider's source is blank, exit.  
If the rider's source's first's target is the return byte, exit.  
If the rider is on any nested double-quote, bump the rider; repeat.  
If the rider's source's first's target is the double-quote byte, bump the rider; exit.  
Repeat.

Handling nested double-quotes requires a little help from this guy, so you should put him back in, too:

To decide if a rider is on any nested double-quote:  
If the rider's source starts with """""", say yes.  
Say no.  
  
72. The "move a rider" routine that handles punctuation marks looks like this:

To move a rider (code rules - mark):  
Bump the rider.

All this routine does is adjust the pointers in the rider so the rider's token spans the mark, and what's left of the rider's source starts just after the mark. All punctuation marks in Plain English are single characters.

73. Symbols are non-letter and non-digit and non-punctuation mark characters that are not considered noise. This the the "move a rider" routine that we use to suck them up:

To move a rider (code rules - symbol):  
Bump the rider.

And this is the routine that decides if a byte is a symbol or not:

To decide if a byte is any symbol:  
If the byte is the caret byte, say yes.  
If the byte is the bar byte, say yes.  
If the byte is the asterisk byte, say yes.  
If the byte is the cross byte, say yes.  
If the byte is the dash byte, say yes.  
If the byte is the slash byte, say yes.  
If the byte is the left-alligator byte, say yes.  
If the byte is the right-alligator byte, say yes.  
If the byte is the left-brace byte, say yes.  
If the byte is the right-brace byte, say yes.  
If the byte is the equal-sign byte, say yes.  
If the byte is 128, say yes.  
If the byte is 130, say yes.  
If the byte is between 132 and 137, say yes.  
If the byte is 139, say yes.  
If the byte is between 145 and 153, say yes.  
If the byte is 155, say yes.  
If the byte is between 161 and 180, say yes.  
If the byte is between 183 and 191, say yes.  
If the byte is 215, say yes.  
If the byte is 247, say yes.  
Say no.

74. Possessives come in two forms: with and without an "s" on the end. An example of the former is "Mary's house"; and example of the latter is "Jesus' Kingdom". This is the "move a rider" routine that deals with both of them:

To move a rider (code rules - possessive):  
Bump the rider.  
If the rider's source is blank, exit.  
If the rider's source starts with "s", bump the rider.

When this routine gets control, we're sitting on a an apostrophe, so we suck it up. If there's no more source, we're done, but if the next character is an "s", we suck that up too.

Simple enough, but we need the 3 deciders below, as well, to peek ahead in the source so we know when to use the above routine:

To decide if a rider is on any possessive:  
If the rider's source is blank, say no.  
If the rider's source's first's target is not the single-quote byte, say no.  
If the rider is on any possessive (type 1 - single-quote, s, non-glom-byte), say yes.  
If the rider is on any possessive (type 2 - preceding s, single-quote, non-glom-byte), say yes.  
Say no.

To decide if a rider is on any possessive (type 1 - single-quote, s, non-glom-byte):  
If the rider's source does not start with "'s", say no.  
Privatize the rider.  
Add 2 to the rider's source's first.  
If the rider's source is blank, say yes.  
If the rider's source's first's target is not any glom byte, say yes.  
Say no.

To decide if a rider is on any possessive (type 2 - preceding s, single-quote, non-glom-byte):  
Privatize the rider.  
If the rider's source's first is the rider's original's first, say no.  
Subtract 1 from the rider's source's first.  
If the rider's source's first's target is not the little-s byte or the big-s byte, say no.  
Add 2 to the rider's source's first.  
If the rider's source is blank, say yes.  
If the rider's source's first's target is not any glom byte, say yes.  
Say no.

Note the "Privatize the rider" statements in the latter two routines. Recall that all parameters in Plain English are passed by reference, not value, so if we change them, they're actually changed. The *privatize* command makes a copy of the specified parameter so we can screw with it without breaking anything above us. It cleverly gives the copy the same name as the original parameter (in this case, "the rider"), while changing the original parameter's name to include the prefix "original" (as in, "the original rider").

75. Anything else in the source code, at this level of detail, is considered a "glom", a collection of characters that we're not quite sure how to handle at this level. This is the "move a rider" routine that sucks up gloms:

To move a rider (code rules - glom):  
Bump the rider.  
If the rider's source is blank, exit.  
If the rider is on any possessive, exit.  
If the rider's source's first's target is any glom byte, repeat.

It needs help from this decider:

To decide if a byte is any glom byte:  
If the byte is any letter, say yes.  
If the byte is any digit, say yes.  
If the byte is the tilde byte, say yes.  
If the byte is the at-sign byte, say yes.  
If the byte is the number-sign byte, say yes.  
If the byte is the percent-sign byte, say yes.  
If the byte is the ampersand byte, say yes.  
If the byte is the underscore byte, say yes.  
If the byte is the single-quote byte, say yes.  
If the byte is the dash byte, say yes.  
If the byte is the cross byte, say yes.  
If the byte is the slash byte, say yes.  
Say no.

And that's it for the bottom-level "move a rider" routines.

76. All of those guys are called by this guy...

To move a rider (code rules):  
Position the rider's token on the rider's source.  
If the rider's source is blank, exit.  
If the rider's source's first's target is noise, move the rider (code rules - noise); exit.  
If the rider's source's first's target is the backslash byte, move the rider (code rules - comment); exit.  
If the rider's source's first's target is the left-bracket byte, move the rider (code rules - remark); exit.  
If the rider's source's first's target is the double-quote byte, move the rider (code rules - string); exit.  
If the rider's source's first's target is the left-paren byte, move the rider (code rules - qualifier); exit.  
If the rider's source's first's target is any mark, move the rider (code rules - mark); exit.  
If the rider is on any possessive, move the rider (code rules - possessive); exit.  
Move the rider (code rules - glom).

...who is not called directly, but who is called by this fellow:

To move a rider returning an error string (code rules):  
Clear the error string.  
Move the rider (code rules).  
If the rider's token is blank, exit.  
If the rider's token's first's target is the left-bracket byte, move the rider returning the error string (code rules - validate remark); exit.  
If the rider's token's first's target is the double-quote byte, move the rider returning the error string (code rules - validate string); exit.  
If the rider's token's first's target is the left-paren byte, move the rider returning the error string (code rules - validate qualifier); exit.

...and who, with a little help from these three...

To move a rider returning an error string (code rules - validate qualifier):  
If the rider's token's length is less than 2, put "Qualifiers need to end with ')'." into the error string; exit.  
If the rider's token's last's target is not the right-paren byte, put "Qualifiers need to end with ')'." into the error string; exit.

To move a rider returning an error string (code rules - validate remark):  
If the rider's token's length is less than 2, put "Remarks need to end with ']'." into the error string; exit.  
If the rider's token's last's target is not the right-bracket byte, put "Remarks need to end with ']'." into the error string; exit.

To move a rider returning an error string (code rules - validate string):  
If the rider's token's length is less than 2, put "Strings need to end with '""'." into the error string; exit.  
If the rider's token's last's target is not the double-quote byte, put "Strings need to end with '""'." into the error string; exit.

...is able to weed out mal-formed remarks, strings, and qualifiers right out of the gate.

77. And that brings us to the last and top-level "move a rider" routine...

To move a rider (compiler rules):  
If the compiler's abort flag is set, clear the rider's token; exit.  
Move the rider returning an error string (code rules).  
If the error string is not blank, abort with the error string and the rider's token's first; exit.  
If the rider's token is blank, exit.  
If the rider's token's first's target is noise, repeat.  
If the rider's token's first's target is the backslash byte, repeat.  
If the rider's token's first's target is the left-bracket byte, repeat.

...who is the guy used by our scanners. We've discussed these "move a rider" routines from the bottom up, as it were, since they're mostly concerned with what impressive people call "lexical analysis" (making words out of letters). We'll be discussing our scanners from the top down since they're mostly concerned with what impressive people call "semantic analysis" (making sentences out of words).

78. We've already put back in the top 2 scanners, the first ("to scan some source files") who just loops through the source files and passes them to the second ("to scan a source file") who simply dispatches the rest of the scanning process to his underlings once he knows what's coming up in the source. These are the deciders he needs to make a good decision:

To decide if a string is the start of any type:  
If the string is "a", say yes.  
If the string is "an", say yes.  
If the string is "some", say yes.  
Say no.

To decide if a string is the start of any global:  
If the string is "the", say yes.  
Say no.

To decide if a string is the start of any routine:  
If the string is "to", say yes.  
Say no.

79. Now the first scanning "underling" is this one:

To scan a type given a rider:  
If the compiler's abort flag is set, exit.  
Add the type to the types given the rider's token's first.  
Move the rider (compiler rules).  
Scan the type's name given the rider.  
Put the type's name into the type's plural name.  
Pluralize the type's plural name.  
If the rider's token is "has" or "have", scan the type given the rider (has or have); exit.  
If the rider's token is not "is" or "are", abort with "I was expecting an 'is' or 'are', but I found '" then the rider's token then "'." and the rider; exit.  
Move the rider (compiler rules).  
If the rider's token is any numeric literal, scan the type given the rider (unit of measure); exit.  
If the rider's token is not any indefinite article, abort with "I was expecting an indefinte article, but I found '" then the rider's token then "'." and the rider; exit.   
Move the rider (compiler rules).  
Scan the type's base name given the rider.  
Scan any optional info for the type given the rider.  
If the rider's token is not the period byte, abort with "Types need to end with a period, I found '" then the rider's token then "'." and the rider; exit.  
Move the rider (compiler rules).  
Index the type.

80. The guy above begins by adding a type to our type list. Then he gets the new type's name, using this general-purpose name-scanning routine:

To scan a name given a rider:  
Clear the name.  
If the compiler's abort flag is set, exit.  
If the rider's token is any name starter, extend the name with the rider's token; move the rider (compiler rules).  
Loop.  
If the compiler's abort flag is set, exit.  
If the rider's token is any name ender, break.  
Extend the name with the rider's token.  
Move the rider (compiler rules).  
Repeat.  
If the name is blank, abort with "I was kind of hoping for a name around here, but I found '" then the rider's token then "'." and the rider; exit.

81. Our name-scanning routine needs a little help from these 2 deciders:

To decide if a string is any name starter:

If the string is any mark, say no.  
If the string is any symbol, say no.  
If the string is any article, say no.  
If the string is any conjunction, say yes.  
If the string is any possessive, say no.  
If the string is any verb, say yes.  
If the string is any operator, say yes.  
If the string is any preposition, say yes.  
If the string is any qualifier, say no.  
If the string is any literal, say no.  
If the string is "called", say yes.  
If the string is "equal", say yes.  
If the string is blank, say no.  
Say yes.

To decide if a string is any name ender:

If the string is any mark, say yes.  
If the string is any symbol, say yes.  
If the string is any article, say yes.  
If the string is any conjunction, say yes.  
If the string is any possessive, say yes.  
If the string is any verb, say yes.  
If the string is any operator, say yes.  
If the string is any preposition, say yes.  
If the string is any qualifier, say yes.  
If the string is any literal, say yes.  
If the string is "called", say yes.  
If the string is "equal", say yes.  
If the string is blank, say yes.

Say no.

82. And those 2 guys need some help from these 14 deciders:

To decide if a string is any mark:  
If the string's length is not 1, say no.  
If the string's first's target is any mark, say yes.  
Say no.

To decide if a string is any symbol:  
If the string's length is not 1, say no.  
If the string's first's target is any symbol, say yes.  
Say no.

To decide if a string is any article:  
If the string is any indefinite article, say yes.  
If the string is "the", say yes.  
Say no.

To decide if a string is any indefinite article:  
If the string is "a", say yes.  
If the string is "an", say yes.  
If the string is "another", say yes.  
If the string is "some", say yes.  
Say no.

To decide if a string is any conjunction:  
If the string is "and", say yes.  
If the string is "both", say yes.  
If the string is "but", say yes.  
If the string is "either", say yes.  
If the string is "neither", say yes.  
If the string is "nor", say yes.  
If the string is "or", say yes.  
Say no.

To decide if a string is any possessive:  
If the string is "'s", say yes.  
If the string is "'", say yes.  
Say no.

To decide if a string is any verb:  
If the string is "are", say yes. \ are not \ aren't  
If the string is "be", say yes.

If the string is "can", say yes. \ cannot \ can't  
If the string is "cannot", say yes.   
If the string is "could", say yes. \ could not \ couldn't  
If the string is "do", say yes. \ do not \ don't  
If the string is "does", say yes. \ does not \ doesn't  
If the string is "is", say yes. \ is not \ isn't  
If the string is "may", say yes. \ may not \ mayn't  
If the string is "should", say yes. \ should not \ shouldn't  
If the string is "was", say yes. \ was not \ wasn't  
If the string is "will", say yes. \ will not \ won't  
If the string is "would", say yes. \ would not \ wouldn't  
If the string is "has", say yes. \ used in type definitions  
If the string is "have", say yes. \ used in type definitions  
If the string ends with "n't", say yes.  
\ trouble makers  
If the string is "begins", say yes. \ does not begin \ doesn't begin  
If the string is "ends", say yes. \ does not end \ doesn't end  
If the string is "looks", say yes. \ does not look \ doesn't look  
If the string is "needs", say yes. \ does not need \ doesn't need  
If the string is "starts", say yes. \ does not start \ doesn't start  
Say no.

To decide if a string is any operator:  
If the string is "plus", say yes.  
If the string is "minus", say yes.  
If the string is "times", say yes.  
If the string is "divided", say yes.  
If the string is "then", say yes.  
Say no.

To decide if a string is any preposition:  
If the string is "about", say yes.  
If the string is "above", say yes.  
If the string is "across", say yes.  
If the string is "after", say yes.  
If the string is "against", say yes.  
If the string is "all", say yes.  
If the string is "alone", say yes.  
If the string is "along", say yes.  
If the string is "among", say yes.  
If the string is "any", say yes.  
If the string is "anywhere", say yes.  
If the string is "around", say yes.  
If the string is "as", say yes.  
If the string is "at", say yes.  
If the string is "away", say yes.  
If the string is "back", say yes.  
If the string is "backward", say yes.  
If the string is "backwards", say yes.  
If the string is "before", say yes.  
If the string is "beginning", say yes.  
If the string is "behind", say yes.  
If the string is "below", say yes.  
If the string is "beneath", say yes.  
If the string is "beside", say yes.  
If the string is "between", say yes.  
If the string is "beyond", say yes.  
If the string is "bigger", say yes.  
If the string is "by", say yes.  
If the string is "close", say yes.  
If the string is "deep", say yes.  
If the string is "diagonally", say yes.  
If the string is "down", say yes.  
If the string is "downward", say yes.  
If the string is "ending", say yes.  
If the string is "everywhere", say yes.  
If the string is "except", say yes.  
If the string is "finishing", say yes.  
If the string is "facing", say yes.  
If the string is "for", say yes.  
If the string is "forward", say yes.  
If the string is "from", say yes.  
If the string is "given", say yes.  
If the string is "giving", say yes.  
If the string is "high", say yes.  
If the string is "in", say yes.  
If the string is "inside", say yes.  
If the string is "into", say yes.  
If the string is "larger", say yes.  
If the string is "leaning", say yes.  
If the string is "left", say yes.  
If the string is "leftward", say yes.  
If the string is "leftways", say yes.  
If the string is "like", say yes.  
If the string is "limiting", say yes.  
If the string is "long", say yes.  
If the string is "near", say yes.  
If the string is "of", say yes.  
If the string is "off", say yes.  
If the string is "on", say yes.  
If the string is "only", say yes.  
If the string is "onto", say yes.  
If the string is "out", say yes.  
If the string is "outside", say yes.  
If the string is "over", say yes.  
If the string is "past", say yes.  
If the string is "pointing", say yes.  
If the string is "returning", say yes.  
If the string is "right", say yes.  
If the string is "rightward", say yes.  
If the string is "rightways", say yes.  
If the string is "shorter", say yes.  
If the string is "sideway", say yes.  
If the string is "sideways", say yes.  
If the string is "similar", say yes.  
If the string is "smaller", say yes.  
If the string is "somewhere", say yes.  
If the string is "slantwise", say yes.  
If the string is "slantway", say yes.  
If the string is "slantways", say yes.  
If the string is "starting", say yes.  
If the string is "tall", say yes.  
If the string is "taller", say yes.  
If the string is "thru", say yes.  
If the string is "through", say yes.  
If the string is "throughout", say yes.  
If the string is "to", say yes.  
If the string is "toward", say yes.  
If the string is "towards", say yes.  
If the string is "under", say yes.  
If the string is "underneath", say yes.  
If the string is "until", say yes.  
If the string is "unto", say yes.  
If the string is "up", say yes.  
If the string is "upward", say yes.  
If the string is "upon", say yes.  
If the string is "using", say yes.  
If the string is "via", say yes.  
If the string is "while", say yes.  
If the string is "wide", say yes.  
If the string is "with", say yes.  
If the string is "within", say yes.  
If the string is "without", say yes.  
Say no.

To decide if a string is any qualifier:  
If the string is blank, say no.  
If the string's first's target is the left-paren byte, say yes.  
Say no.

To decide if a string is any literal:  
If the string is any pointer literal, say yes.  
If the string is any numeric literal, say yes.  
If the string is any string literal, say yes.  
If the string is any flag literal, say yes.  
If the string is any hex literal, say yes.  
Say no.

To decide if a string is any pointer literal:  
If the string is "nil", say yes.  
Say no.

To decide if a string is any string literal:  
If the string is blank, say no.  
If the string's first's target is the double-quote byte, say yes.  
Say no.

To decide if a string is any hex literal:  
If the string is blank, say no.  
If the string's first's target is the dollar-sign byte, say yes.  
Say no.

And now you know why primitive programming languages don't allow spaces in names.

83. As we collect the words that make up a name, we're going to need the routine below, to put back the spaces that were lost in the tokenizing process:

To extend a string with another string:  
If the other string is blank, exit.  
If the string is not blank, append the space byte to the string.  
Append the other string to the string.

84. We also need these 3 guys, who figure out the plural version of a name:

To pluralize a string: \ nouns only  
If the string is "alumnus", put "alumni" into the string; exit.  
If the string is "auto", put "autos" into the string; exit.  
If the string is "cello", put "cellos" into the string; exit.  
If the string is "dwarf", put "dwarfs" into the string; exit.  
If the string is "foot", put "feet" into the string; exit.  
If the string is "genus", put "genera" into the string; exit.  
If the string is "goose", put "geese" into the string; exit.  
If the string is "hippo", put "hippos" into the string; exit.  
If the string is "louse", put "lice" into the string; exit.  
If the string is "memo", put "memos" into the string; exit.  
If the string is "mouse", put "mice" into the string; exit.  
If the string is "ox", put "oxen" into the string; exit.  
If the string is "phenomenon", put "phenomena" into the string; exit.  
If the string is "photo", put "photos" into the string; exit.  
If the string is "phylum", put "phyla" into the string; exit.  
If the string is "piano", put "pianos" into the string; exit.  
If the string is "pimento", put "pimentos" into the string; exit.  
If the string is "pro", put "pros" into the string; exit.  
If the string is "proof", put "proofs" into the string; exit.  
If the string is "radius", put "radii" into the string; exit.  
If the string is "rhinoceros", put "rhinoceri" into the string; exit.  
If the string is "roof", put "roofs" into the string; exit.  
If the string is "solo", put "solos" into the string; exit.  
If the string is "soprano", put "sopranos" into the string; exit.  
If the string is "staff", put "staffs" into the string; exit.  
If the string is "tooth", put "teeth" into the string; exit.  
If the string is "torus", put "tori" into the string; exit.  
If the string is "turf", put "turfs" into the string; exit.  
If the string is "virus", put "viruses" into the string; exit.  
If the string ends with "sh", append "es" to the string; exit.  
If the string ends with "ch", append "es" to the string; exit.  
If the string ends with "man", remove the last two bytes from the string; append "en" to the string; exit.  
If the string ends with "child", append "ren" to the string; exit.  
If the string ends with "ex", remove the last two bytes from the string; append "ices" to the string; exit. \ or add ES  
If the string ends with "fe", remove the last two bytes from the string; append "ves" to the string; exit.  
If the string ends with "is", remove the last two bytes from the string; append "es" to the string; exit.  
If the string ends with "ix", remove the last two bytes from the string; append "ices" to the string; exit. \ or add ES  
If the string ends with "ma", append "ta" to the string; exit. \ or add S  
If the string ends with any consonant and "y", remove the last byte from the string; append "ies" to the string; exit.  
If the string ends with any vowel and "o", append "s" to the string; exit.  
If the string ends with any vowel and "y", append "s" to the string; exit.  
If the string ends with "f", remove the last byte from the string; append "ves" to the string; exit.  
If the string ends with "s", append "es" to the string; exit.  
If the string ends with "x", append "es" to the string; exit.  
If the string ends with "z", append "es" to the string; exit.  
Append "s" to the string.

To decide if a string ends with any consonant and another string:  
Slap a substring on the string.  
Subtract the other string's length from the substring's last.  
If the substring is blank, say no.  
If the substring's last's target is not any consonant, say no.  
If the string does not end with the other string, say no.  
Say yes.

To decide if a string ends with any vowel and another string:  
Slap a substring on the string.  
Subtract the other string's length from the substring's last.  
If the substring is blank, say no.  
If the substring's last's target is not any vowel, say no.  
If the string does not end with the other string, say no.  
Say yes.

85. And now that we've got our names straight, we can get back to scanning our types. Types come in different forms; simple types, like "An address is a number," are typically just subsets of other types. But types that have the word "has" or "have" after the type name, like "A button has a box and a name," are record types. To scan record types, we need this routine:

To scan a type given a rider (has or have):  
If the compiler's abort flag is set, exit.  
Move the rider (compiler rules).  
Put "record" into the type's base name.  
Scan the type's fields given the rider.  
If the rider's token is not the period byte, abort with "Types need to end with a period, I found '" then the rider's token then "'." and the rider; exit.  
Move the rider (compiler rules).  
Index the type.

86. Records have fields so we need some field-processing routines, too (keep in mind that fields behave like, and are treated like variables, in many places:

To scan some fields given a rider:  
If the compiler's abort flag is set, exit.  
Add a field to the fields given "field" and the rider's token's first.  
Scan the field given the rider.  
If the rider's token is not any pause, exit.  
Scan any pauses given the rider.  
Repeat.

To add a variable to some variables given a kind and a locus:  
Create the variable given the kind.  
Append the variable to the variables.  
Put the locus into the variable's locus.

To scan a field given a rider:  
If the compiler's abort flag is set, exit.  
Scan the field given the rider (type part).  
Loop.  
If the compiler's abort flag is set, exit.  
If the rider's token is "called", scan the field given the rider (called part); repeat.  
If the rider's token is "at", scan the field given the rider (redefine part); repeat.  
If the rider's token is "(reference)", scan the field given the rider (reference part); repeat.

87. As you can see from the routine above, each field in a record has a type of its own, and may have a variety of other special features. We handle those with these 6 scanners:

To scan a field given a rider (type part):  
If the compiler's abort flag is set, exit.  
If the rider's token is any numeric literal, scan the field given the rider (type part - byte array); exit.  
Scan the field given the rider (type part - normal).

To scan a field given a rider (type part - normal):  
If the compiler's abort flag is set, exit.  
If the rider's token is not any indefinite article, abort with "I was expecting an indefinite article, but found '" then the rider's token then "'." and the rider; exit.  
If the rider's token is "another", put "other" into the field's name.  
Move the rider (compiler rules).  
Scan a name given the rider.  
Extend the field's name with the name.

To scan a field given a rider (called part):  
If the compiler's abort flag is set, exit.  
Move the rider (compiler rules).  
If the field's type name is blank, put the field's name into the field's type name. \ byte array type name already filled in  
Scan the field's name given the rider.

To scan a field given a rider (redefine part):  
If the compiler's abort flag is set, exit.  
Move the rider (compiler rules).  
If the rider's token is not "the", abort with "I was expecting the word 'the', but found '" then the rider's token then "'." and the rider; exit.  
Move the rider (compiler rules).  
Scan the field's redefinition target name given the rider.

To scan a field given a rider (reference part):  
If the compiler's abort flag is set, exit.  
Move the rider (compiler rules).  
Set the field's reference flag.

To scan a field given a rider (type part - byte array):  
If the compiler's abort flag is set, exit.  
Convert the rider's token to a ratio.  
Reduce the ratio.  
If the ratio's denominator is not 1, abort with "You have to specify a whole number of bytes for a byte array" and the rider; exit.  
Put the ratio's numerator into the field's count.  
Move the rider (compiler rules).  
If the rider's token is not "byte" or "bytes", abort with "Nothing but the word 'bytes' is appropriate after a counted field designator." and the rider; exit.  
Move the rider (compiler rules).  
Put "byte" into the field's type name.

88. Since fields in a record can be separated by a variety of things (called *pauses*), we need these 2 routines, too:

To scan any pauses given a rider:  
If the compiler's abort flag is set, exit.  
If the rider's token is not any pause, exit.  
Move the rider (compiler rules).  
Repeat.

To decide if a string is any pause:  
If the string is ",", say yes.  
If the string is ";", say yes.  
If the string is "and", say yes.  
If the string is "or", say yes.  
Say no.

89. Some types, like "An inch is 1440 twips", are "unit of measure" types. The CAL uses such types to automatically convert units of measure at compile time. These types are scanned by this guy:

To scan a type given a rider (unit of measure):  
If the compiler's abort flag is set, exit.  
Convert the rider's token to the type's scale.  
If the type's scale is 0, abort with "Zero is an invalid scale." and the type's locus; exit.  
Move the rider (compiler rules).  
Scan the type's base name given the rider.  
If the rider's token is not the period byte, abort with "Types need to end with a period, I found '" then the rider's token then "'." and the rider; exit.  
Move the rider (compiler rules).  
Index the type.

90. The 4 routines below are used to scan any optional information that might appear at the end of a type definition (like the target type of a pointer type, or fields that extend another record type):

To decide if a string is the start of any optional info:  
If the string is "to", say yes.  
If the string is "with", say yes.  
Say no.

To scan any optional info for a type given a rider:  
If the compiler's abort flag is set, exit.  
If the rider's token is "to", scan any optional info for the type given the rider (pointer); exit.  
If the rider's token is "with", scan any optional info for the type given the rider (record); exit.

To scan any optional info for a type given a rider (pointer):  
Move the rider (compiler rules).  
If the rider's token is not any indefinite article, abort with "The 'to' here needs to be followed by an indefinite article, not '" then the rider's token then "'." and the rider; exit.   
Move the rider (compiler rules).  
Scan the type's target name given the rider.

To scan any optional info for a type given a rider (record):  
Move the rider (compiler rules).  
Scan the type's fields given the rider.

And that's it for scanning types.

91. Now we need some routines to scan our globals. This is the top-level guy and 3 of his helpers:

To scan a global given a rider:  
If the compiler's abort flag is set, exit.  
Add the global to the globals given "global" and the rider's token's first.  
Move the rider (compiler rules).  
Scan the global's name given the rider.  
If the rider's token is "has" or "have", scan the global given the rider (has or have); exit.  
If the rider's token is not "is" or "are", abort with "Geez. I need an 'is' or an 'are' hereabouts, but all I found was '" then the rider's token then "'." and the rider; exit.  
Move the rider (compiler rules).  
If the rider's token is not any indefinite article, scan the global given the rider (literal term); exit.  
Move the rider (compiler rules).  
Scan the global's type name given the rider.    
If the rider's token is the start of any optional info, scan the global given the rider (optional info); exit.  
If the rider's token is "equal", scan the global given the rider (data part); exit.  
If the rider's token is not the period byte, abort with "All global definitions must end with a period. Including this one." and the rider; exit.   
Move the rider (compiler rules).  
Index the global.

To add a variable to some variables given a kind and a locus:  
Create the variable given the kind.  
Append the variable to the variables.  
Put the locus into the variable's locus.

To create a variable given a kind:  
Allocate memory for the variable.  
Put the kind into the variable's kind.  
Put 1 into the variable's count.

To index a global:  
If the compiler's abort flag is set, exit.  
If the global is nil, exit.  
If the global's name is in the global index, abort with "'" then the global's name then "' is a duplicate global variable." and the global's locus; exit.  
Index the global given the global's name and the global index.

92. There's a bit of redundancy here because global variables can include implied, "inline" type defintions. So this guy should look somewhat familiar:

To scan a global given a rider (has or have):  
If the compiler's abort flag is set, exit.  
Move the rider (compiler rules).  
Put "~inline " then the global's name then " type" into a name.  
Put the name then "s" into a plural name.  
Add a type to the types given the name and the plural name and "record" and the global's locus.  
Index the type.  
Put the type's name into the global's type name.  
Scan the type's fields given the rider.  
If the rider's token is not the period byte, abort with "Your punctuation is bad here." and the rider; exit.   
Move the rider (compiler rules).  
Index the global.

93. And this guy should look familiar, too:

To scan a global given a rider (optional info):  
If the compiler's abort flag is set, exit.  
Put "~inline " then the global's name then " type" into a name.  
Put the name then "s" into a plural name.  
Add a type to the types given the name and the plural name and the global's type name and the global's locus.  
Index the type.  
Put the type's name into the global's type name.  
Scan any optional info for the type given the rider.  
If the rider's token is not the period byte, abort with "All global definitions must end with a period. Including this one." and the rider; exit.   
Move the rider (compiler rules).  
Index the global.

94. Some global variables have initial values, so we need this routine:

To scan a global given a rider (data part):  
If the compiler's abort flag is set, exit.  
Move the rider (compiler rules).  
If the rider's token is not "to", abort with "I need the word 'to' after the word 'equal', not '" then the rider's token then "'." and the rider; exit.   
Move the rider (compiler rules).  
Scan the global's global body given the rider.  
If the rider's token is not the period byte, abort with "All global definitions must end with a period. Including this one." and the rider; exit.   
Move the rider (compiler rules).  
Index the global.

We're not quite ready to deal with those initial values, so we just suck up the "body" of the global definition and keep it for later. This is the routine that does that:

To scan a global body given a rider:  
If the compiler's abort flag is set, exit.  
If the rider's token is blank, exit.  
Put the rider's token's first into the global body's first.  
Put -1 into the global body's last.  
Loop.  
If the compiler's abort flag is set, exit.  
If the rider's token is blank, exit.  
If the rider's token is the colon byte, exit.  
If the rider's token is the period byte, exit.  
Put the rider's token's last into the global body's last.  
Move the rider (compiler rules).  
Repeat.

A similar thing occurs here:

To scan a global given a rider (literal term):  
If the compiler's abort flag is set, exit.  
Scan the global's global body given the rider.  
If the rider's token is not the period byte, abort with "All global definitions must end with a period. Including this one." and the rider; exit.   
Move the rider (compiler rules).  
Index the global.

And that's it for globals, for now.

95. This is our top-level routine scanner, with a couple of his trivial helpers:

To scan a routine given a rider:  
If the compiler's abort flag is set, exit.  
Add the routine to the routines given the rider's token's first.  
Move the rider (compiler rules).  
Scan the routine's routine header given the rider.  
If the rider's token is the semi-colon byte, scan the routine given the rider (alternate wording); exit.

If the rider's token is not the colon byte, abort with "This routine header is improperly formulated." and the routine's locus; exit.  
Move the rider (compiler rules).  
Scan the routine's routine body given the rider.

To add a routine to some routines given a locus:  
Create the routine.  
Append the routine to the routines.  
Put the locus into the routine's locus.

To create a routine:  
Allocate memory for the routine.

96. It's too soon to do much with routines, so we just collect each routine's header and body into a substring and save them for later:

To scan a routine header given a rider:  
If the compiler's abort flag is set, exit.  
If the rider's token is blank, exit.  
Put the rider's token's first into the routine header's first.  
Put -1 into the routine header's last.  
Loop.  
If the compiler's abort flag is set, exit.  
If the rider's token is blank, exit.  
If the rider's token is the semi-colon byte, exit.

If the rider's token is the colon byte, exit.  
If the rider's token is the period byte, exit.  
Put the rider's token's last into the routine header's last.  
Move the rider (compiler rules).  
Repeat.

To scan a routine body given a rider:  
If the compiler's abort flag is set, exit.  
If the rider's token is blank, exit.  
If the rider's token is the start of any definition, exit.  
Put the rider's token's first into the routine body's first.  
Loop.  
If the compiler's abort flag is set, exit.  
If the rider's token is blank, exit.  
Put the rider's token's last into the routine body's last.  
If the rider's token is the colon byte, abort with "You may have forgotten a period somewhere above this. You better check." and the rider; exit.  
If the rider's token is not the period byte, move the rider (compiler rules); repeat.  
Move the rider (compiler rules).  
If the rider's token is the start of any definition, exit.  
Repeat.

97. There's some special treatment required for routines with alternate headers:

To scan a routine given a rider (alternate wording):

Privatize the rider.  
Move the rider (compiler rules).  
If the rider's token is not "to", abort with "I think that there semi-colon is misplaced." and the original rider; exit.  
Put the original rider's token's first into the routine's routine body's first.  
Put the original rider's token's last into the routine's routine body's last.  
Move the original rider (compiler rules).

98. And we need this little guy so we can find the end of a routine body. Routine bodies end with the start of the next type, global or routine definition:

To decide if a string is the start of any definition:  
If the string is "to", say yes.  
If the string is "a", say yes.  
If the string is "an", say yes.  
If the string is "the", say yes.  
If the string is "some", say yes.  
Say no.

99. And now we should be able to run. CTRL-R. Do it. When you're in the blue, find the "CAL-4700's son's daughter" directory, open it, open the source files, and do a list (CTRL-L).

100. Still in the blue, on a free tab, open the daughter's ".lst" file. Now we've got something to look at -- almost 12,000 lines, in fact. Let's examine just a few.

You'll notice that almost all the types, except for "byte" and "record", have -1 (hex FFFFFFFF) in their length field, like this one:

/type/abort message/abort messages/FFFFFFFF/string////0/

That's because we haven't yet calculated those lengths.

You'll also notice that "implied" types have a tilde on the front of their names, like this:

/type/~inline compiler type/~inline compiler types/FFFFFFFF/record////0/

Thirdly, you'll notice that record types have their fields listed as variables, like so:

/type/entry/entries/FFFFFFFF/thing////0/  
/variable/field/no/name address////00000000/no/1/no////  
/variable/field/no/thunk address////00000000/no/1/no////  
/variable/field/no/name////00000000/no/1/no////  
/variable/field/no/address////00000000/no/1/no////

If you skip down to the next section of the listing (just find "GLOBALS:" with no quotes and a colon on the end), you'll see lots of globals, with their types filled in, but with zeros in their addresses, like this:

/variable/global/no/adding built-in memory routines timer//timer//00000000/no/1/no////  
/variable/global/no/adding built-in startup routine timer//timer//00000000/no/1/no////  
/variable/global/no/addressing timer//timer//00000000/no/1/no////  
/variable/global/no/break tag//tag//00000000/no/1/no////  
/variable/global/no/calculating timer//timer//00000000/no/1/no////

In the "LITERALS:" section, you'll find nothing at all, because we haven't done anything with literals yet (except save them in global and routine body substrings).

In the "ROUTINES:" section, you'll see we've got a lot of routines hanging on the list, but that we know almost nothing about them at this point. That's because all we did was squirrel away their headers and bodies in substrings for later.

Further down you can see that our "TYPE INDEX:" is now getting some use, with some entries sharing buckets:

TYPE INDEX:

/367 buckets/405 refers/

Our "GLOBAL INDEX:" is also getting some use:

GLOBAL INDEX:

/558 buckets/729 refers/

Our other indexes are empty:

LITERAL INDEX:

ROUTINE INDEX:

UTILITY INDEX:

Our "IMPORTS:" section is also empty, since we haven't compiled any references to DLLs yet:

IMPORTS:

Finally, at the very bottom, you'll see that we've got all six source files properly loaded (as before)...

SOURCE FILES:

/source file/c:\cal-4700's son's daughter\the compiler/  
/source file/c:\cal-4700's son's daughter\the desktop/  
/source file/c:\cal-4700's son's daughter\the editor/  
/source file/c:\cal-4700's son's daughter\the finder/  
/source file/c:\cal-4700's son's daughter\the noodle/  
/source file/c:\cal-4700's son's daughter\the writer/

...and that scanning now accounts for the bulk of our compile time:

TIMERS:

/timer/loading/0/  
/timer/scanning/437/  
/timer/resolving types/0/  
/timer/resolving globals/0/  
/timer/compiling routine headers/0/  
/timer/calculating/0/  
/timer/adding built-in memory routines/0/  
/timer/indexing utilities/0/  
/timer/compiling routine bodies/0/  
/timer/adding built-in startup routine/0/  
/timer/offsetting/0/  
/timer/addressing/0/  
/timer/transmogrifying/0/  
/timer/linking/0/  
/timer/writing/0/  
/timer/total/562/

So far, so good. Next up, resolution of our types.

---

Okay, you know the drill.

101. In the gray, open the Son of CAL's compiler, move our temporary abort line down a notch, and uncomment the "resolve the types" step:

To compile a directory:  
Compile the directory (start).  
Compile the directory (load the source files).  
Compile the directory (scan the source files).  
Compile the directory (resolve the types).  
Abort with "I'm not old enough to make babies.".  
\Compile the directory (resolve the globals).  
\Compile the directory (compile the headers of the routines).  
\Compile the directory (calculate lengths and offsets of types).  
\Compile the directory (add the built-in memory routines).  
\Compile the directory (index the routines for utility use).  
\Compile the directory (compile the bodies of the routines).  
\Compile the directory (add and compile the built-in startup routine).  
\Compile the directory (offset parameters and variables).  
\Compile the directory (address).  
\Compile the directory (transmogrify).  
\Compile the directory (link).  
\Compile the directory (write the exe).  
\Compile the directory (stop).

102. In this step we're going to whip through all of our types four times, looking for different things on each pass:

To compile a directory (resolve the types):  
If the compiler's abort flag is set, exit.  
Show status "Resolving...".  
Start the resolving types timer.  
Resolve the types (expand things).  
Resolve the types (index plurals).  
Resolve the types (base types).  
Resolve the types (optional info).  
Stop the resolving types timer.

103: First Pass: (a) make a new record type with previous and next pointers on the front for every type defined as a "thing"; make the original type look like a pointer to the new record type; and add the implied types needed for easy list making as well:

To resolve some types (expand things):  
If the compiler's abort flag is set, exit.  
Get a type from the types.  
If the type is nil, exit.  
Resolve the type (expand thing).  
Repeat.

To resolve a type (expand thing): \ type's base type is not resolved yet  
If the compiler's abort flag is set, exit.  
If the type is nil, exit.  
If the type's base type is not nil, exit.  
If the type's name is "thing", exit.  
If the type cannot be reduced to "thing" using the base name, exit.  
Find a base type given the type's base name and the type index.  
\  add thing record  
Put the type's name then " record" into a name.  
Put the type's name then " records" into a plural name.  
Put the base type's name then " record" into a base name.  
Add a record type to the types given the name and the plural name and the base name.  
Add a field to the record type's fields given "next " then the type's name and "next" and the type's name and yes.  
Add another field to the record type's fields given "previous " then the type's name and "previous" and the type's name and yes.  
If the type's fields are not empty, append the type's fields to the record type's fields.  
Index the record type.  
\ fix up original type to look like a pointer  
Put the record type's name into the type's target name.  
\ add chain type  
Add a chain type to the types given the type's plural name and "" and the base type's plural name.  
Add a third field to the chain type's fields given "first " then the type's name and "first" and the type's name and yes.  
Add a fourth field to the chain type's fields given "last " then the type's name and "last" and the type's name and yes.  
Index the chain type.

104. We're going to need this decider to do all that:

To decide if a type can be reduced to a string using the base name:  
If the type is nil, say no.  
If the type's name is the string, say yes.  
If the type's cooking flag is set, say no.  
Set the type's cooking flag.  
Find a base type given the type's base name and the type index.  
If the base type is nil, clear the type's cooking flag; say no.  
If the base type can be reduced to the string using the base name, clear the type's cooking flag; say yes.  
Clear the type's cooking flag.  
Say no.

105. And we're going to need this little helper as well:

To add a field to some fields given a name and a nickname and a type name and a reference flag:  
Add the field to the fields given "field" and nil.  
Put the name into the field's name.  
Put the nickname into the field's nickname.  
Put the type name into the field's type name.  
Put the reference flag into the field's reference flag.

106: Second Pass: Make sure all the as-yet unindexed plural names make it into the type index:

To resolve some types (index plurals):  
If the compiler's abort flag is set, exit.  
Get a type from the types.  
If the type is nil, exit.  
Resolve the type (index plural).  
Repeat.

To resolve a type (index plural):  
If the compiler's abort flag is set, exit.  
If the type is nil, exit.  
If the type's plural name is blank, exit.  
If the type can be reduced to "thing" using the base name, exit.  
Index the type given the type's plural name.

107: Third Pass: Make sure the base types for all higher-level types are solid:

To resolve some types (base types):  
If the compiler's abort flag is set, exit.  
Get a type from the types.  
If the type is nil, exit.  
Resolve the type (base type).  
Repeat.

To resolve a type (base type):  
If the compiler's abort flag is set, exit.  
If the type is nil, exit.  
If the type's base type is not nil, exit.  
Find the type's base type given the type's base name and the type index.  
If the type's base type is nil, abort with "Invalid base type '" then the type's base name then "'." and the type's locus; exit.  
Set the type's cooking flag.  
If the type's base type's cooking flag is set, abort with "Recursive definition with '" then the type's base name then "'." and the type's locus; exit.  
Resolve the type's base type (base type).  
Clear the type's cooking flag.

We use the "cooking flag" to make sure we don't run into ourselves when we're chasing down types. It's okay, for example, to have a *nickname* be a *name* and a *name* be a *string*, but you can't have a *nickname* be a *name* and a *name* be a *nickname*.

108. Fourth Pass: See what we can do with all the optional information our types might have tacked on to them:

To resolve some types (optional info):  
If the compiler's abort flag is set, exit.  
Get a type from the types.  
If the type is nil, exit.  
Resolve the type (optional info).  
Repeat.

To resolve a type (optional info):  
If the compiler's abort flag is set, exit.  
If the type is nil, exit.  
If the type's optional info resolved flag is set, exit.  
If the type's base type is the type, exit. \ for built in types  
Check for invalid optional info on the type.  
Set the type's cooking flag.  
Resolve the type's base type (optional info).  
If the type can be reduced to "pointer", resolve the type (optional info - pointer).  
If the type can be reduced to "record", resolve the type (optional info - record).  
If the type can be reduced to "number", resolve the type (optional info - scale).  
Clear the type's cooking flag.  
Set the type's optional info resolved flag.

109. At this point, we need lots of helpers; these, for starters:

To check for invalid optional info on a type:  
If the compiler's abort flag is set, exit.  
If the type is nil, exit.  
If the type's target name is not blank, check for invalid optional info on the type (target).  
If the type's fields are not empty, check for invalid optional info on the type (fields).  
If the type's scale is not 0, check for invalid optional info on the type (scale).

To check for invalid optional info on a type (fields):  
If the compiler's abort flag is set, exit.  
If the type cannot be reduced to "record", abort with "Optional 'with' info is only allowed on record types." and the type's locus; exit.

To check for invalid optional info on a type (scale):  
If the compiler's abort flag is set, exit.  
If the type cannot be reduced to "number", abort with "Scales are only allowed on number types." and the type's locus; exit.

To check for invalid optional info on a type (target):  
If the compiler's abort flag is set, exit.  
If the type cannot be reduced to "pointer", abort with "Optional 'to' info is only allowed on pointer types." and the type's locus; exit.  
  
110: These helpers are needed too:

To resolve a type (optional info - pointer):  
If the compiler's abort flag is set, exit.  
If the type's target type is not nil, exit.  
If the type's target name is blank, put the type's base type's target type into the type's target type; exit.  
Find the type's target type given the type's target name and the type index.  
If the type's target type is nil, abort with "I can't quite see what you're pointing at. Please clarify." and the type's locus; exit.

To resolve a type (optional info - record):  
If the compiler's abort flag is set, exit.  
If the type's fields are empty, copy the type's base type's fields into the type's fields; exit.  
Loop.  
If the compiler's abort flag is set, exit.  
Get a field from the type's fields (backwards).  
If the field is nil, break.  
If the field is duplicated in the type's fields, abort with "The '" then the field's name then "' field is defined more than once." and the field's locus; exit.  
Resolve the field.  
If the compiler's abort flag is set, exit.  
If the field's type's cooking flag is set, abort with "Recursive definition with '" then the field's type's name then "'." and the field's locus; exit.  
Resolve the field's type (optional info).  
If the field's reference flag is not set, repeat.  
If the field's type cannot be reduced to "pointer", abort with "Exactly what do you mean by '(reference)' here on this '" then the field's name then "'?" and the field's locus; exit.  
Repeat.  
Eliminate duplicate nicknames from the type's fields.  
If the type cannot be reduced to "thing record", exit.  
If the type's fields' count is greater than 2, exit.  
Copy the type's base type's fields into the type's fields (data fields only).

To resolve a type (optional info - scale):  
If the compiler's abort flag is set, exit.  
Put the type's base type into a base type.  
If the type's scale is 0, put the type's base type's scale into the type's scale; exit.  
If the base type's scale is 0, exit.  
Multiply the type's scale by the base type's scale.  
Put the base type's base type into the type's base type.

111. And these two deciders should be put back in:

To decide if a type can be reduced to a type name:  
If the type is nil, say no.  
Privatize the type.  
Loop.  
If the type is nil, say no.  
If the type's name is the type name, say yes.  
If the type's base type is the type, say no.  
Put the type's base type into the type.  
Repeat.

To decide if a variable is duplicated in some variables:  
Get another variable from the variables.  
If the other variable is nil, say no.  
If the other variable's name is blank, repeat.  
If the other variable is the variable, repeat.  
If the other variable's name is the variable's name, say yes.  
Repeat.

That last decider is used to eliminate duplicate fields in records (recall that fields are considered variables).

112. When a record type is extended, we need to duplicate fields, so we're going to need these guys, too:

To copy a field into another field:  
If the field is nil, void the other field; exit.  
Allocate memory for the other field.  
Put the field's locus into the other field's locus.  
Put the field's name into the other field's name.  
Put the field's nickname into the other field's nickname.  
Put the field's type name into the other field's type name.  
Put the field's type into the other field's type.  
Put the field's count into the other field's count.  
Put the field's redefinition target name into the other field's redefinition target name.  
Put the field's reference flag into the other field's reference flag.  
Put the field's offset into the other field's offset.

To copy some fields into some other fields:  
Get a field from the fields.  
If the field is nil, exit.  
Copy the field into another field.  
Append the other field to the other fields.  
Repeat.

To copy some fields into some other fields (data fields only):  
If the fields' first is nil, exit.  
If the fields' first's next is nil, exit.  
Put the fields' first's next's next into a field.  
Loop.  
If the field is nil, exit.  
Copy the field into another field.  
Append the other field to the other fields.  
Get the field from the fields.  
Repeat.

113. We resolve our *global* variables later, but we resolve our "field variables" on this pass through the types:

To resolve a field:  
If the compiler's abort flag is set, exit.  
If the field is nil, exit.  
Resolve the field as a variable.  
If the field's nickname is not any valid field name, clear the field's nickname.  
If the field's name is not any valid field name, abort with "'" then the field's name then "' is not a valid field name." and the field's locus; exit.

To resolve a variable:  
If the compiler's abort flag is set, exit.  
If the variable is nil, exit.  
If the variable's type is not nil, exit.  
If the variable's type name is not blank, resolve the variable (explicit type name); exit.  
Find the variable's type and the variable's nickname given the variable's name.  
If the variable's type is not nil, put the variable's type's name into the variable's type name; exit.  
Abort with "I can't find the '" then the variable's name then "' type." and the variable's locus.

To resolve a variable (explicit type name):  
Find the variable's type given the variable's type name and the type index.  
If the variable's type is nil, abort with "I can't find the '" then the variable's type name then "' type." and the variable's locus; exit.

114. Perhaps you noticed various calls to "find" routines in the stuff above. These are the ones we should put back now:

To find a type and a nickname given a name:  
Void the type.  
Clear the nickname.  
Slap a substring on the name.  
Loop.  
If the substring is blank, exit.  
Find the type given the substring and the type index.  
If the type is not nil, break.  
Skip to the next word in the substring.  
Repeat.  
Put the name's first into another substring's first.  
Put the substring's first minus 2 into the other substring's last.  
If the other substring is not blank, put the other substring into the nickname.  
If the other substring is blank, put the substring into the nickname.

To find a variable given a name:  
If the current routine is nil, void the variable; exit.  
Find the variable given the name and the current routine's locals.  
If the variable is not nil, exit.  
Find the variable given the name and the current routine's parameters.  
If the variable is not nil, exit.  
Find the variable given the name and the global index.  
If the variable is not nil, exit.  
Find the variable given the name and the literal index.

To find a variable given a name and some variables:  
Void the variable.  
Loop.  
Get the variable from the variables.  
If the variable is nil, exit.  
If the variable's name is the name, exit.  
If the variable's nickname is the name, exit.  
Repeat.

115. We're going to need this little guy to help us with nickname processing:

To skip to the next word in a substring:  
If the substring is blank, exit.  
If the substring's first's target is noise, break.  
Add 1 to the substring's first.  
Repeat.  
Skip any leading noise in the substring.

116. Plus this guy, to make sure the programmer doesn't misuse our three reserved field names:

To decide if a name is any valid field name:  
If the name is "magnitude", say no.  
If the name is "target", say no.  
If the name is "whereabouts", say no.  
Say yes.

117. And these 3 guys to help us eliminate duplicate nicknames:

To eliminate duplicate nicknames from some variables:  
Create an index given 101.  
Eliminate duplicate nicknames from the variables given the index.  
Destroy the index.

To eliminate duplicate nicknames from some variables given an index:  
If the index is nil, exit.  
Loop.  
Get a variable from the variables.  
If the variable is nil, exit.  
Eliminate duplicate nicknames given the variable and the index.  
Repeat.

To eliminate duplicate nicknames given a variable and an index:  
If the variable is nil, exit.  
If the index is nil, exit.  
Find another variable given the variable's nickname and the index.  
If the other variable is nil, find the other variable given the variable's name and the index.  
If the other variable is nil, index the variable given the variable's nickname and the index; exit.  
Clear the other variable's nickname.  
Clear the variable's nickname.

118. And that's it; types resolved. You know the drill. Run it; when you're in the blue, open the daughter and list it. Our types won't be complete yet, but will be a bit more robust and error-free.

Here's a notable example. In the Noodle, a wyrd is defined as a record like this:

A wyrd has a low byte and a high byte.

And a number is defined as a record like this:

A number has  
a first byte,  
a second byte,  
a third byte,  
a fourth byte,  
a low wyrd at the first byte,  
a high wyrd at the third byte.

We define wyrds and numbers in this way so we can easily get to their pieces.

You probably remember that in the compiler we have this type:

A tag is a number.

And now that our types have been resolved, you can see in the listing that all the fields in the number type have been copied into the tag type (remember that fields are treated like variables):

/type/tag/tags/FFFFFFFF/number/number///0/  
/variable//no/first byte/first/byte/byte/00000000/no/1/no////  
/variable//no/second byte/second/byte/byte/00000000/no/1/no////  
/variable//no/third byte/third/byte/byte/00000000/no/1/no////  
/variable//no/fourth byte/fourth/byte/byte/00000000/no/1/no////  
/variable//no/low wyrd/low/wyrd/wyrd/00000000/no/1/no/first byte///  
/variable//no/high wyrd/high/wyrd/wyrd/00000000/no/1/no/third byte///

And here's another notable example. In the compiler we define a "source file" like this:

A source file is a thing with a path and a buffer.

But now that our types have been resolved, that definition has been modified and expanded to include (a) a "source file" type that is nothing but a pointer to a "source file record"; (b) a "source file record" that has "next source file" and "previous source file" pointers on the front for easy chaining; and (c) a compiler-generated type, "source files" (plural) with "first source file" and "last source file" fields that can be used as the anchor for a list. Here they are in the listing:

/type/source file/source files/FFFFFFFF/thing/thing/source file record/source file record/0/  
/variable//no///byte/byte/00000000/no/4/no////

/type/source file record/source file records/FFFFFFFF/thing record/thing record///0/  
/variable/field/no/next source file/next/source file/source file/00000000/no/1/yes////  
/variable/field/no/previous source file/previous/source file/source file/00000000/no/1/yes////  
/variable/field/no/path/path/path/path/00000000/no/1/no////  
/variable/field/no/buffer/buffer/buffer/buffer/00000000/no/1/no////

/type/source files//FFFFFFFF/things/things///0/  
/variable/field/no/first source file/first/source file/source file/00000000/no/1/yes////  
/variable/field/no/last source file/last/source file/source file/00000000/no/1/yes////

A third notable example is the "foot" unit-of-measure type, which was defined like this:

A foot is 12 inches.

Which refers to this definition of an inch:

An inch is 1440 twips.

Which refers to this definition of a twip:

A twip is a number.

Now that our types have been resolved, we find that the "foot" type looks like this:

/type/foot/feet/FFFFFFFF/inches/twip///17280/  
/variable//no/first byte/first/byte/byte/00000000/no/1/no////  
/variable//no/second byte/second/byte/byte/00000000/no/1/no////  
/variable//no/third byte/third/byte/byte/00000000/no/1/no////  
/variable//no/fourth byte/fourth/byte/byte/00000000/no/1/no////  
/variable//no/low wyrd/low/wyrd/wyrd/00000000/no/1/no/first byte///  
/variable//no/high wyrd/high/wyrd/wyrd/00000000/no/1/no/third byte///

Note that the "foot" type not only has an appropriate plural form ("feet"), but that it also has all the fields that any other number has, and that the "scale factor" (last item on the first line above) is 17280, which is 12 times 1440, or the number of twips in an foot. The "scale factor" is used, at compile time, to convert literals in routine calls that have a specified unit of measure to units expected by the called routine. When, for example, in the Cal Monet we said:

Resize the picture to 5-1/2 inches by 5-1/2 inches.

The CAL automatically passed 7920 to the resize routine (who was expecting twips, not inches). And had we made the picture just a little bigger by saying...

Resize the picture to 1/2 foot by 1/2 foot.

The CAL would have passed 8640 to the resize routine (8640 is the number of twips in 6 inches, or 1/2 foot).

Looking further in our listing, we ca see that our type index is a little busier:

TYPE INDEX:

/840 buckets/1001 refers/

And that all that resolving of types took almost no time at all:

TIMERS:

/timer/loading/62/  
/timer/scanning/453/  
/timer/resolving types/47/  
/timer/resolving globals/0/  
/timer/compiling routine headers/0/  
/timer/calculating/0/  
/timer/adding built-in memory routines/0/  
/timer/indexing utilities/0/  
/timer/compiling routine bodies/0/  
/timer/adding built-in startup routine/0/  
/timer/offsetting/0/  
/timer/addressing/0/  
/timer/transmogrifying/0/  
/timer/linking/0/  
/timer/writing/0/  
/timer/total/719/

Next up: Resolving global variables.

---

119. Moving on. Back in the gray, adjust the "compile a directory" routine to look like this:

To compile a directory:  
Compile the directory (start).  
Compile the directory (load the source files).  
Compile the directory (scan the source files).  
Compile the directory (resolve the types).  
Compile the directory (resolve the globals).  
Abort with "I'm not old enough to make babies.".  
\Compile the directory (compile the headers of the routines).  
\Compile the directory (calculate lengths and offsets of types).  
\Compile the directory (add the built-in memory routines).  
\Compile the directory (index the routines for utility use).  
\Compile the directory (compile the bodies of the routines).  
\Compile the directory (add and compile the built-in startup routine).  
\Compile the directory (offset parameters and variables).  
\Compile the directory (address).  
\Compile the directory (transmogrify).  
\Compile the directory (link).  
\Compile the directory (write the exe).  
\Compile the directory (stop).

120. Here are the top-level routines for this step:

To compile a directory (resolve the globals):  
If the compiler's abort flag is set, exit.  
Show status "Resolving...".  
Start the resolving globals timer.  
Resolve the globals.  
Stop the resolving globals timer.

To resolve some globals:  
If the compiler's abort flag is set, exit.  
Get a global from the globals.  
If the global is nil, exit.  
Resolve the global.  
Repeat.

121. And here's the gist of it:

To resolve a global:  
If the global is nil, exit.  
If the compiler's abort flag is set, exit.  
If the global's global body is not blank, resolve the global (compile body).  
If the global's type name is blank, abort with "Invalid global '" then the global's name then "'." and the global's locus; exit.  
Resolve the global as a variable.

To resolve a global (compile body):  
If the global is nil, exit.  
If the compiler's abort flag is set, exit.  
Slap a rider on the global's global body.  
Move the rider (compiler rules).  
If the rider's token is not any literal, abort with "I need a literal here, but I found '" then the rider's token then "'." and the rider; exit.  
Compile a literal given the rider.  
If the compiler's abort flag is set, exit.  
If the rider's token is not blank, abort with "Is a period missing around here?" and the rider; exit.  
If the global's type name is blank, put the literal's type's name into the global's type name.  
Put the literal into the global's literal.  
Index the literal.

122. We use the word "compile" here (instead of just "scan" or "resolve") since we're actually getting to the place where we are formatting some of the information in the source code exactly as it will appear in the executable file. Specifically, literals used to initialize global variables:

To compile a literal given a rider:  
If the compiler's abort flag is set, exit.  
Add the literal to the literals given the rider's token's first.  
If the rider's token is any flag literal, compile the literal given the rider (flag); exit.  
If the rider's token is any hex literal, compile the literal given the rider (hex); exit.  
If the rider's token is any numeric literal, compile the literal given the rider (numeric); exit.  
If the rider's token is any pointer literal, compile the literal given the rider (pointer); exit.  
If the rider's token is any string literal, compile the literal given the rider (string); exit.  
Abort with "Internal error - compile a term given a rider (literal)" and the rider.

123. Literals don't have names, but we give them names anyway to make them easier to use and to spot (not to mention, chase around) in the listing file. Since the names are generated by the compiler, we put a tilde as the first character:

To generate a name given a string:  
Put the string into the name.  
Add 1 to the compiler's name count.  
Convert the compiler's name count to another string.  
Append the other string to the name.

To add a literal to some variables given a locus:  
Add the literal as a variable to the variables given "literal" and the locus.  
Generate the literal's name given "~L".

124. Several types of literals have to be converted to "executable ready" format:

To compile a literal given a rider (flag):  
If the compiler's abort flag is set, exit.  
Convert the rider's token to a flag.  
Convert the flag to the literal's data.  
Move the rider (compiler rules).  
Put "flag" into the literal's type name.  
Resolve the literal.

To compile a literal given a rider (hex):  
If the compiler's abort flag is set, exit.  
Put the rider's token into a nibble substring.  
Add 1 to the nibble substring's first.  
Convert the nibble substring to the literal's data.  
Move the rider (compiler rules).  
Put "hex string" into the literal's type name.  
Resolve the literal.

To compile a literal given a rider (numeric - integer):  
If the compiler's abort flag is set, exit.  
Convert the rider's token to a number.  
Convert the number to the literal's data.  
Move the rider (compiler rules).  
Put "number" into the literal's type name.  
Resolve the literal.

To compile a literal given a rider (numeric - ratio):  
If the compiler's abort flag is set, exit.  
Convert the rider's token to a ratio.  
Convert the ratio to the literal's data.  
Move the rider (compiler rules).  
Put "ratio" into the literal's type name.  
Resolve the literal.

To compile a literal given a rider (numeric):  
If the compiler's abort flag is set, exit.  
If the rider is followed by any unit of measure, compile the literal given the rider (numeric - with unit of measure); exit.  
If the rider's token is any integer literal, compile the literal given the rider (numeric - integer); exit.  
If the rider's token is any ratio literal, compile the literal given the rider (numeric - ratio); exit.  
If the rider's token is any mixed literal, compile the literal given the rider (numeric - ratio); exit.

To compile a literal given a rider (numeric - with unit of measure):  
If the compiler's abort flag is set, exit.  
Convert the rider's token to a ratio.  
Move the rider (compiler rules).  
Scan a name given the rider.  
Find a type given the name and the type index.  
If the type is nil, abort with "'" then the name then "' is an invalid unit of measure." and the literal's locus; exit.  
If the type cannot be reduced to "number", abort with "'" then the name then "' is an invalid unit of measure." and the literal's locus; exit.  
Put the type's scale into a final ratio.  
If the type's scale is 0, put 1/1 into the final ratio.  
Multiply the final ratio by the ratio.  
Reduce the final ratio.  
If the final ratio's denominator is 0, abort with "We seem to have an invalid unit of measure here." and the literal's locus.  
Put the final ratio's numerator divided by the final ratio's denominator into a number.  
Put the type's name into the literal's type name.  
Convert the number to the literal's data.  
Resolve the literal.

To compile a literal given a rider (pointer):  
If the compiler's abort flag is set, exit.  
Convert nil to the literal's data.  
Move the rider (compiler rules).  
Put "pointer" into the literal's type name.  
Resolve the literal.

To compile a literal given a rider (string):  
If the compiler's abort flag is set, exit.  
Put the rider's token into the literal's data.  
Unquote the literal's data.  
Move the rider (compiler rules).  
Put "string" into the literal's type name.  
Resolve the literal.

125. We'll be needing this decider to peek ahead a little so we can spot units-of-measure that follow numeric literals (like the "inches" part of "3 inches":

To decide if a rider is followed by any unit of measure:  
Privatize the rider.  
Move the rider (compiler rules).  
If the rider's token is not any name starter, say no.  
Scan a name given the rider.  
Find a type given the name and the type index.  
If the type is nil, say no.  
If the name is not "times", say yes. \ special because times is an infix operator and a unit of measure  
If the rider's token is the start of any expression, say no.  
Say yes.

126. The decider above is also used when we're parsing expressions (later on), so we need this guy to go with him:

To decide if a string is the start of any expression:  
If the string is any sign, say yes.  
If the string is any indefinite article, say yes.  
If the string is "the", say yes.  
If the string is any literal, say yes.  
Say no.

127. We index our literals so we can find them, quickly, later when they're needed:

To index a literal:  
If the compiler's abort flag is set, exit.  
If the literal is nil, exit.  
If the literal's name is in the literal index, abort with "Internal error - index a literal" and the literal's locus; exit.  
Index the literal given the literal's name and the literal index.

128. The various literal conversion routines are generally useful, so we keep them in the Noodle. In one case, however, we need an "alternate wording" for the conversion routine, just to make sure calls get dispatched to the right place:

To convert a nibble substring to a string: employ convert a nibble string to a hex string.

129. And that's it for resolving globals! Time to see what all that does to the listing. Run it. In the blue, open up the daughter's source files and list. The open up the updated listing file. Jump to the "GLOBALS:" Section and you'll see that initialized globals are now associated with named literals, like so:

GLOBALS:

/variable/global/no/adding built-in memory routines timer//timer/timer/00000000/no/1/no////  
/variable/global/no/adding built-in startup routine timer//timer/timer/00000000/no/1/no////  
/variable/global/no/addressing timer//timer/timer/00000000/no/1/no////  
/variable/global/no/break tag//tag/tag/00000000/no/1/no//~L1//  
/variable/global/no/calculating timer//timer/timer/00000000/no/1/no////  
/variable/global/no/call external tag//tag/tag/00000000/no/1/no//~L2//  
/variable/global/no/call indirect tag//tag/tag/00000000/no/1/no//~L3//  
/variable/global/no/call internal tag//tag/tag/00000000/no/1/no//~L4//

Note the "~L1" and "~L2" and similar literal names near the end of the tag globals above. Now jump to the "LITERALS:" Section and you'll see those very literals with their values, backways, ready for inclusion in our executable file:

LITERALS:

/variable/literal/no/~L1//number/number/00000000/no/1/no///01000000/  
/variable/literal/no/~L2//number/number/00000000/no/1/no///02000000/  
/variable/literal/no/~L3//number/number/00000000/no/1/no///03000000/  
/variable/literal/no/~L4//number/number/00000000/no/1/no///04000000/

If you look for "~L113" you'll find a very long line in the listing that starts like this:

/variable/literal/no/~L113//hex string/hex string/00000000/no/1/no///524946463A020...

The source for that literal is in the Noodle as part of the definition of our "cluck" sound global variable:

The cluck sound is a wave equal to $524946463A020...

Later we'll see how the compiler generates code to copy literal values like these into the corresponding global variables just before he calls the programmer's "run" routine.

More literals, and the code to copy them into the right places, will be added later as we stumble on literals in routine bodies.

Let's check our timers and call it a day:

TIMERS:

/timer/loading/94/  
/timer/scanning/437/  
/timer/resolving types/31/  
/timer/resolving globals/94/  
/timer/compiling routine headers/0/  
/timer/calculating/0/  
/timer/adding built-in memory routines/0/  
/timer/indexing utilities/0/  
/timer/compiling routine bodies/0/  
/timer/adding built-in startup routine/0/  
/timer/offsetting/0/  
/timer/addressing/0/  
/timer/transmogrifying/0/  
/timer/linking/0/  
/timer/writing/0/  
/timer/total/937/

Looks like manipulating all those huge hex literals in the Noodle cost us a few milliseconds. The cluck sound is only one of them; the Osmosian font is another, and it's almost 72,000 nibbles long. I thnk we can live with 94 milliseconds in any case.

---

130. Onward. In the gray, move our temporary abort line down, and uncomment the next step.

To compile a directory:  
Compile the directory (start).  
Compile the directory (load the source files).  
Compile the directory (scan the source files).  
Compile the directory (resolve the types).  
Compile the directory (resolve the globals).  
Compile the directory (compile the headers of the routines).  
Abort with "I'm not old enough to make babies.".  
\Compile the directory (calculate lengths and offsets of types).  
\Compile the directory (add the built-in memory routines).  
\Compile the directory (index the routines for utility use).  
\Compile the directory (compile the bodies of the routines).  
\Compile the directory (add and compile the built-in startup routine).  
\Compile the directory (offset parameters and variables).  
\Compile the directory (address).  
\Compile the directory (transmogrify).  
\Compile the directory (link).  
\Compile the directory (write the exe).  
\Compile the directory (stop).

131. Our goal in this step is to gather enough information so we can formulate a convenient "moniker" for each routine. Let's begin by putting back the definition of a moniker (and a partial moniker, while we're at it):

A moniker is a string.

A partial moniker is a moniker.

Then let's consider an example from the Cal Monet. The moniker for this routine...

To create a work given a URL:

... looks like this:

create [work] from/given/with/using [url]

It consists of 4 "monikettes": the string "create", the parameter type "[work]", the string "from/given/with/using", and the parameter type "[url]".

132. We should uncomment the monikette thing now...

A monikette is a thing with  
a string,  
a type (reference),  
a variable (reference),  
\ for bubbling  
a current type (reference),  
a current substring.

A monikette's *string* is used to hold string-type monikettes (like "create" or "from/given/with/using"), while a monikette's *type* points to the type definition of a parameter. A monikette's *variable* isn't used until later, when we parse expressions.

A routine record has a list of monikettes and a list of parameters in it. As we compile a routine header, we hang the routine's parameters on the parameter list, and the monikettes (both string and parameter types) on the monikette list.

133. We should put back these 3 helpers at this time:

To create a monikette:  
Allocate memory for the monikette.

To add a monikette to some monikettes given a string:  
Create the monikette.  
Append the monikette to the monikettes.  
\ from/given/with/using  
If the string is "from", put "from/given/with/using" into the monikette's string; exit.  
If the string is "given", put "from/given/with/using" into the monikette's string; exit.  
If the string is "with", put "from/given/with/using" into the monikette's string; exit.  
If the string is "using", put "from/given/with/using" into the monikette's string; exit.  
\ in/into/to  
If the string is "in", put "in/into/to" into the monikette's string; exit.  
If the string is "into", put "in/into/to" into the monikette's string; exit.  
If the string is "to", put "in/into/to" into the monikette's string; exit.  
\ is/are  
If the string is "are", put "is/are" into the monikette's string; exit.  
If the string is "is", put "is/are" into the monikette's string; exit.  
\ aren't/isn't  
If the string is "isn't", put "is/aren't" into the monikette's string; exit.  
If the string is "aren't", put "is/aren't" into the monikette's string; exit.  
\ backward/backwards/counterclockwise/counter-clockwise/anticlockwise/anti-clockwise  
If the string is "backward", put "backward" into the monikette's string; exit.  
If the string is "backwards", put "backward" into the monikette's string; exit.  
If the string is "counterclockwise", put "backward" into the monikette's string; exit.  
If the string is "counter-clockwise", put "backward" into the monikette's string; exit.  
If the string is "anticlockwise", put "backward" into the monikette's string; exit.  
If the string is "anti-clockwise", put "backward" into the monikette's string; exit.  
\ at/on  
If the string is "at", put "at/on" into the monikette's string; exit.  
If the string is "on", put "at/on" into the monikette's string; exit.  
\ other  
Put the string into the monikette's string.

To add a monikette to some monikettes given a type:  
Create the monikette.  
Append the monikette to the monikettes.   
Put the type into the monikette's type.

134. We convert them into a moniker, using the routines below. The conversion routine is in two parts in case we don't want to convert the whole monikette list at one time.

To convert some monikettes to a moniker:  
Convert the monikettes to the moniker given the monikettes' last.

To convert some monikettes to a moniker given a monikette:  
Clear the moniker.  
Loop.  
Get a current monikette from the monikettes.  
If the current monikette is nil, exit.  
Append the current monikette to the moniker.  
If the current monikette is the monikette, exit.  
Repeat.

To append a monikette to a moniker:  
If the monikette is nil, exit.  
If the moniker is not blank, append the space byte to the moniker.  
If the monikette's type is not nil, append "[" and the monikette's type's name and "]" to the moniker (fast); exit.  
If the monikette's string is not blank, append the monikette's string to the moniker; exit.

135. Now that we've got all that out of the way, let's compile some headers. These are the top-level routines:

To compile a directory (compile the headers of the routines):  
If the compiler's abort flag is set, exit.  
Show status "Compiling...".  
Start the compiling routine headers timer.  
Compile the headers of the routines.  
Stop the compiling routine headers  timer.

To compile the headers of some routines:  
If the compiler's abort flag is set, exit.  
Get a routine from the routines.  
If the routine is nil, exit.  
Compile the header of the routine.  
Repeat.

To compile the header of a routine:  
If the routine is nil, exit.  
If the compiler's abort flag is set, exit.  
Slap a rider on the routine's routine header.  
Move the rider (compiler rules).  
If the rider's token is "decide", compile the header of the routine given the rider (decider); exit.  
If the rider is the start of any function, compile the header of the routine given the rider (function); exit.  
If the rider's token is "compatibly", compile the header of the routine given the rider (callback); exit.  
Compile the header of the routine given the rider (procedure).

136. And these are the helpers, for each kind of routine (decider, function (the tricky one), callback, and procedure):

To decide if a rider is the start of any function:  
If the compiler's abort flag is set, say no.  
If the rider's token is not "put", say no.  
Privatize the rider.  
Move the rider (compiler rules).  
If the rider's token is not any article, say no.  
Move the rider (compiler rules).  
Scan a name given the rider.  
If the rider's token is any possessive, say yes.

To compile some monikettes given a rider:   
If the compiler's abort flag is set, exit.  
If the rider's token is any monikette string, add a monikette to the monikettes given the rider's token; move the rider (compiler rules); repeat.  
If the rider's token is not the start of any expression, exit.  
Compile an expression given the rider.  
If the compiler's abort flag is set, exit.  
Add another monikette to the monikettes given the expression.  
Repeat.

To compile the header of a routine given a rider (decider):  
If the compiler's abort flag is set, exit.  
Move the rider (compiler rules).  
If the rider's token is not "if", abort with "It's 'Decide IF.' We always say, 'Decide IF'. Got it?" and the rider; exit.  
Move the rider (compiler rules).  
Set the routine's decider flag.  
Compile the routine's monikettes and the routine's parameters given the rider.  
If any of the routine's monikettes are negative words, abort with "It's a bad idea to use negatives in decider names." and the rider; exit.  
Convert the routine's monikettes to the routine's moniker.  
If the routine's moniker is in the routine index, abort with "I already know how to '" then the routine's moniker then "'." and the routine's locus; exit.  
Index the routine given the routine's monikettes and the routine index.

To compile the header of a routine given a rider (function):  
If the compiler's abort flag is set, exit.  
Set the routine's function flag.  
Add a monikette to the routine's monikettes given "put".  
Move the rider (compiler rules).  
If the rider's token is any indefinite article, set a flag.  
If the flag is set, compile the routine's monikettes and the routine's parameters given the rider (indefinite article).  
If the flag is not set, compile the routine's monikettes and the routine's parameters given the rider (definite article).  
If the rider's token is not any possessive, abort with "An 's is missing or misplaced here." and the rider; exit.  
Move the rider (compiler rules).  
Scan a name given the rider.  
Add another monikette to the routine's monikettes given "'s " then the name.  
If the rider's token is not "into", abort with "I was expecting the word 'into', but found '" then the rider's token then "'." and the rider; exit.  
Add a third monikette to the routine's monikettes given "into".  
Convert the routine's monikettes to a moniker.  
If the moniker is in the routine index, abort with "I already know how to '" then the moniker then "'." and the routine's locus; exit.  
Index the routine given the routine's monikettes and the routine index.  
Move the rider (compiler rules).  
If the rider's token is not any indefinite article, abort with "I was expecting an indefinite article, but found '" then the rider's token then "'." and the rider; exit.  
Compile the routine's monikettes and the routine's parameters given the rider (indefinite article).  
If the rider's token is not blank, abort with "There's extra stuff on the end of this function." and the rider; exit.  
Convert the routine's monikettes to the routine's moniker.  
Index the routine given the routine's monikettes and the routine index. \ for employs

To compile the header of a routine given a rider (callback):  
If the compiler's abort flag is set, exit.  
Move the rider (compiler rules).  
Set the routine's callback flag.  
Compile the routine's monikettes and the routine's parameters given the rider.  
Convert the routine's monikettes to the routine's moniker.  
If the routine's moniker is in the routine index, abort with "I already know how to '" then the routine's moniker then "'." and the routine's locus; exit.  
Index the routine given the routine's monikettes and the routine index.

To compile the header of a routine given a rider (procedure):  
If the compiler's abort flag is set, exit.  
Compile the routine's monikettes and the routine's parameters given the rider.  
Convert the routine's monikettes to the routine's moniker.  
If the routine's moniker is in the routine index, abort with "I already know how to '" then the routine's moniker then "'." and the routine's locus; exit.  
Index the routine given the routine's monikettes and the routine index.

137. And these are the guys that separate the parameters from the rest of the "blah, blah, blah" in a routine header:

To decide if a string is any monikette string:  
If the string is blank, say no.  
If the string is "the", say no.  
If the string is any indefinite article, say no.  
If the string is any possessive, say no.  
If the string is any literal, say no.  
If the string is any mark, say no.  
If the string is any symbol, say no.  
Say yes.

To compile some monikettes and some parameters given a rider:  
If the compiler's abort flag is set, exit.  
If the rider's token is blank, exit.  
If the rider's token is "the", compile the monikettes and the parameters given the rider (definite article); repeat.  
If the rider's token is any indefinite article, compile the monikettes and the parameters given the rider (indefinite article); repeat.  
If the rider's token is any monikette string, compile the monikettes and the parameters given the rider (monikette string); repeat.  
Abort with "'" then the rider's token then "' is not valid in a routine header." and the rider.

To compile some monikettes and some parameters given a rider (definite article):  
If the compiler's abort flag is set, exit.  
Move the rider (compiler rules).  
Scan a name given the rider.  
If the compiler's abort flag is set, exit.  
Prepend "the " to the name.  
Add a monikette to the monikettes given the name.

To compile some monikettes and some parameters given a rider (indefinite article):  
If the compiler's abort flag is set, exit.  
Add a parameter to the parameters given "parameter" and the rider's token's first.  
If the rider's token is "another", put "other" into the parameter's name.  
Move the rider (compiler rules).  
Scan a name given the rider.  
Extend the parameter's name with the name.  
If the rider's token is "called", compile the call'd part given the rider and the parameter.  
Resolve the parameter.  
If the parameter is duplicated in the parameters, abort with "Dude, you already have a parameter called '" then the parameter's name then "'." and the parameter's locus; exit.  
Add a monikette to the monikettes given the parameter's type.

To compile some monikettes and some parameters given a rider (monikette string):  
If the compiler's abort flag is set, exit.  
Add a monikette to the monikettes given the rider's token.  
Move the rider (compiler rules).

138. This special guy handles "renamed parameters" in a routine header (like "a number called x"):

To compile the call'd part given a rider and a variable:  
If the compiler's abort flag is set, exit.  
If the variable is nil, exit.  
Put the variable's name into the variable's type name.  
Move the rider (compiler rules).  
Scan the variable's name given the rider.  
  
139. And these three little guys are used to spot "negative words" in decider headers:

To decide if any of some monikettes are negative words:  
Get a monikette from the monikettes.  
If the monikette is nil, say no.  
If the monikette's string is any negative word, say yes.  
Repeat.

To decide if a string is any monikette string:  
If the string is blank, say no.  
If the string is "the", say no.  
If the string is any indefinite article, say no.  
If the string is any possessive, say no.  
If the string is any literal, say no.  
If the string is any mark, say no.  
If the string is any symbol, say no.  
Say yes.

To decide if a string is any negative word:  
If the string is "not", say yes.  
If the string is "cannot", say yes.  
If the string is "nothing", say yes.  
If the string ends with "n't", say yes.  
Say no.

When a decider is written with a "positive" header, the CAL will understand both the positive and negative forms. For example, "To decide if a string is blank" will automatically serve as "to decide if a string is *not* blank".

140. Since there's a lot of string concatenation required in monikette-to-moniker conversions, and since we do those a lot, we added in this helper to make those concatenations faster than usual:

To append a string and another string and a third string to a fourth string (fast):  
Put the fourth string's length into a length.  
Add the string's length to the length.  
Add the other string's length to the length.  
Add the third string's length to the length.  
Reassign a pointer given the length.  
Put the pointer into a byte pointer.  
Copy bytes from the fourth string's first to the byte pointer for the fourth string's length.  
Add the fourth string's length to the byte pointer.  
Copy bytes from the string's first to the byte pointer for the string's length.  
Add the string's length to the byte pointer.  
Copy bytes from the other string's first to the byte pointer for the other string's length.  
Add the other string's length to the byte pointer.  
Copy bytes from the third string's first to the byte pointer for the third string's length.  
Unassign the fourth string's first.  
Put the pointer into the fourth string's first.  
Put the pointer plus the length minus 1 into the fourth string's last.

141. And these are the routines we use to index routines:

To index a routine given some monikettes and an index:  
If the routine is nil, exit.  
Clear a partial moniker.  
Loop.  
Get a monikette from the monikettes.  
If the monikette is nil, exit.  
Append the monikette to the partial moniker.  
If the monikette is the monikettes' last, break.  
Index the partial moniker given the index (special).  
Repeat.  
Index the routine given the partial moniker as a moniker and the index (special).

To index a routine given a moniker and an index (special):  
Find a refer given the moniker and the index.  
If the refer is not nil, put the routine into the refer's pointer; exit.  
Index the routine given the moniker and the index.

To index a partial moniker given an index (special):  
Find a refer given the partial moniker and the index.  
If the refer is not nil, exit.  
Index the partial moniker in the index.

To decide if a moniker is in an index:   
Find a refer given the moniker and the index.  
If the refer is nil, say no.  
If the refer's pointer is nil, say no.  
Say yes.

142. Ready to run. Do it. In the blue, open up and list the daughter. Then open up the listing. Nice work! Now we've got monikers and parameter lists for each and every routine; here's a sampling:

ROUTINES:

/routine/abort from/given/with/using [string]/no/no/no/no//0/0/00000000/  
/variable/parameter/no/string/string/string/string/00000000/no/1/no////

/routine/abort from/given/with/using [string] and [byte pointer]/no/no/no/no//0/0/00000000/  
/variable/parameter/no/string/string/string/string/00000000/no/1/no////  
/variable/parameter/no/byte pointer/byte pointer/byte pointer/byte pointer/00000000/no/1/no////

/routine/abort from/given/with/using [string] and [rider]/no/no/no/no//0/0/00000000/  
/variable/parameter/no/string/string/string/string/00000000/no/1/no////  
/variable/parameter/no/rider/rider/rider/rider/00000000/no/1/no////

And you can see our "ROUTINE INDEX:" is getting a workout, too:

ROUTINE INDEX:

/4881 buckets/7862 refers/

/bucket/

/refer/make [ratio pair] from/given/with/using.../

And that it was somewhat expensive, time-wise, to do all that:

TIMERS:

/timer/loading/78/  
/timer/scanning/422/  
/timer/resolving types/32/  
/timer/resolving globals/109/  
/timer/compiling routine headers/266/  
/timer/calculating/0/  
/timer/adding built-in memory routines/0/  
/timer/indexing utilities/0/  
/timer/compiling routine bodies/0/  
/timer/adding built-in startup routine/0/  
/timer/offsetting/0/  
/timer/addressing/0/  
/timer/transmogrifying/0/  
/timer/linking/0/  
/timer/writing/0/  
/timer/total/1328/

Time to move on.

---

143. Time for an easy step. In the gray, move our temporary abort line down another notch and uncomment the line above it:

To compile a directory:  
Compile the directory (start).  
Compile the directory (load the source files).  
Compile the directory (scan the source files).  
Compile the directory (resolve the types).  
Compile the directory (resolve the globals).  
Compile the directory (compile the headers of the routines).  
Compile the directory (calculate lengths and offsets of types).  
Abort with "I'm not old enough to make babies.".  
\Compile the directory (add the built-in memory routines).  
\Compile the directory (index the routines for utility use).  
\Compile the directory (compile the bodies of the routines).  
\Compile the directory (add and compile the built-in startup routine).  
\Compile the directory (offset parameters and variables).  
\Compile the directory (address).  
\Compile the directory (transmogrify).  
\Compile the directory (link).  
\Compile the directory (write the exe).  
\Compile the directory (stop).

144. Here's the top-level routine:

To compile a directory (calculate lengths and offsets of types):  
If the compiler's abort flag is set, exit.  
Show status "Calculating...".  
Start the calculating timer.  
Calculate the lengths of the types.  
Calculate the offsets in the types.  
Calculate the redefinition offsets in the types.  
Stop the calculating timer.

145. And here are the first couple helpers:

To calculate the lengths of some types:  
If the compiler's abort flag is set, exit.  
Get a type from the types.  
If the type is nil, exit.  
Calculate the length of the type.  
Repeat.

To calculate the length of a type:  
If the type is nil, exit.  
If the type's length is not -1, exit.  
Calculate the length of the type's base type.  
If the type's fields are empty, put the type's base type's length into the type's length; exit.  
Put 0 into the type's length.  
Loop.  
If the compiler's abort flag is set, exit.  
Get a field from the type's fields.  
If the field is nil, exit.  
If the field's redefinition target name is not blank, repeat.  
Calculate the length of the field's type.  
Add the field's type's length times the field's count to the type's length.  
Repeat.

146. And here are some more helpers:

To calculate the offsets in a type:  
If the type is nil, exit.  
Put 0 into an offset.  
Loop.  
If the compiler's abort flag is set, exit.  
Get a field from the type's fields.  
If the field is nil, exit.  
If the field's redefinition target name is not blank, repeat.  
Put the offset into the field's offset.  
Add the field's type's length times the field's count to the offset.  
Repeat.

To calculate the offsets in some types:  
If the compiler's abort flag is set, exit.  
Get a type from the types.  
If the type is nil, exit.  
Calculate the offsets in the type.  
Repeat.

147. And here are even more helpers:

To calculate the redefinition offsets in a type:  
If the type is nil, exit.  
Loop.  
If the compiler's abort flag is set, exit.  
Get a field from the type's fields.  
If the field is nil, exit.  
If the field's redefinition target name is blank, repeat.  
Find another field given the field's redefinition target name and the type's fields.  
If the other field is nil, abort with "Invalid redefine. I can't find the '" then the field's redefinition target name then "' field." and the field's locus; exit.  
Put the other field's offset into the field's offset.  
Repeat.

To calculate the redefinition offsets in some types:  
If the compiler's abort flag is set, exit.  
Get a type from the types.  
If the type is nil, exit.  
Calculate the redefinition offsets in the type.  
Repeat.

148. Done! Run it. In the blue, open up and list the daughter. Then open the updated listing and consider the first few types.

You can see that bytes are still 1 byte long:

/type/byte/bytes/00000001/byte/byte///0/

And that our prototype record is still 0 bytes long.

/type/record/records/00000000/record/record///0/

But our abort message, which is a string, is now 8 bytes long, with a first byte pointer at offset 0, and a last byte pointer at offset 4:

/type/abort message/abort messages/00000008/string/string///0/  
/variable//no/first byte pointer/first/byte pointer/byte pointer/00000000/no/1/no////  
/variable//no/last byte pointer/last/byte pointer/byte pointer/00000004/no/1/no////

You can also see that our "address" type has some redefinitions in it:

/type/address/addresses/00000004/number/number///0/  
/variable//no/first byte/first/byte/byte/00000000/no/1/no////  
/variable//no/second byte/second/byte/byte/00000001/no/1/no////  
/variable//no/third byte/third/byte/byte/00000002/no/1/no////  
/variable//no/fourth byte/fourth/byte/byte/00000003/no/1/no////  
/variable//no/low wyrd/low/wyrd/wyrd/00000000/no/1/no/first byte///  
/variable//no/high wyrd/high/wyrd/wyrd/00000002/no/1/no/third byte///

An address is a number, 4 bytes in length overall. The low wyrd redefines the first two bytes and is thus at the same offset as the first byte; the high wyrd redefines the last two bytes, and is thus at the same offset as the third byte.

Finally, a glance at our timers shows that this step, important as it is, took almost no time at all:

TIMERS:

/timer/loading/109/  
/timer/scanning/453/  
/timer/resolving types/16/  
/timer/resolving globals/109/  
/timer/compiling routine headers/266/  
/timer/calculating/15/  
/timer/adding built-in memory routines/0/  
/timer/indexing utilities/0/  
/timer/compiling routine bodies/0/  
/timer/adding built-in startup routine/0/  
/timer/offsetting/0/  
/timer/addressing/0/  
/timer/transmogrifying/0/  
/timer/linking/0/  
/timer/writing/0/  
/timer/total/1390/

Sweet!

---

149. Onward. In the gray, set things up to "add the built-in memory routines":

To compile a directory:  
Compile the directory (start).  
Compile the directory (load the source files).  
Compile the directory (scan the source files).  
Compile the directory (resolve the types).  
Compile the directory (resolve the globals).  
Compile the directory (compile the headers of the routines).  
Compile the directory (calculate lengths and offsets of types).  
Compile the directory (add the built-in memory routines).  
Abort with "I'm not old enough to make babies.".  
\Compile the directory (index the routines for utility use).  
\Compile the directory (compile the bodies of the routines).  
\Compile the directory (add and compile the built-in startup routine).  
\Compile the directory (offset parameters and variables).  
\Compile the directory (address).  
\Compile the directory (transmogrify).  
\Compile the directory (link).  
\Compile the directory (write the exe).  
\Compile the directory (stop).

We saw earlier (in the "resolve the types" step) that when a programmer defines a dynamic record type using the "is a thing" syntax, the CAL does a little behind-the-scenes type-creation and type-manipulation to keep "things" simple at the programmer's level. Case in point. This type definition...

A source file is a thing with a path and a buffer.

...is modified and expanded to include (a) a "source file" type that is nothing but a pointer to a "source file record"; (b) a "source file record" that has the same fields as the programmer specified plus "next source file" and "previous source file" pointers on the front for easy chaining; and (c) a compiler-generated type, "source files" (plural) with "first source file" and "last source file" fields that can be used as the anchor for a list. Here they are in the listing:

/type/source file/source files/FFFFFFFF/thing/thing/source file record/source file record/0/  
/variable//no///byte/byte/00000000/no/4/no////

/type/source file record/source file records/FFFFFFFF/thing record/thing record///0/  
/variable/field/no/next source file/next/source file/source file/00000000/no/1/yes////  
/variable/field/no/previous source file/previous/source file/source file/00000000/no/1/yes////  
/variable/field/no/path/path/path/path/00000000/no/1/no////  
/variable/field/no/buffer/buffer/buffer/buffer/00000000/no/1/no////

/type/source files//FFFFFFFF/things/things///0/  
/variable/field/no/first source file/first/source file/source file/00000000/no/1/yes////  
/variable/field/no/last source file/last/source file/source file/00000000/no/1/yes////

Now we also want the programmer to be able to program *as if* he took the time to write routines with these headers:

To allocate memory for a source file:

To destroy a source file:

To destroy some source files:

So we have the compiler add those routines *as if* they were in the programmer's source code, in this step.

150. These are the upper-level routines that make that happen:

To compile a directory (add the built-in memory routines):  
If the compiler's abort flag is set, exit.  
Show status "Adding...".  
Start the adding built-in memory routines timer.  
Add the allocate and deallocate and finalize and destroy routines.  
Stop the adding built-in memory routines timer.

To add the allocate and deallocate and finalize and destroy routines:  
If the compiler's abort flag is set, exit.  
Get a type from the types.  
If the type is nil, exit.  
Add the finalize routine for the type.  
Add the allocate routine for the type.  
Add the deallocate routine for the type.  
Add the destroy routine for the type.  
Repeat.

151. We begin with the "finalize" routines, that the programmer never sees. These are used to clean up any string memory that may have been automatically allocated on the Heap. And these are the routines that add those routines to the routine list (as if we had found them in the source code):

To add the finalize routine for a type:  
If the type is nil, exit.  
If the compiler's abort flag is set, exit.  
If the type should not be finalized, exit.  
If the type's name is "string", add the finalize routine for the type (string); exit.  
If the type can be reduced to "string", exit. \ prevents generation of finalizer for derived string types  
Add the finalize routine for the type (record).

To add the finalize routine for a type (string):  
If the type is nil, exit.  
If the compiler's abort flag is set, exit.  
Add a routine to the routines given nil.  
Append "~finalize a " then the type's name to the routine's header string.  
Append "intel $50. " to the routine's body string. \ push eax  
Append "unassign the string's first. " to the routine's body string.  
Append "intel $58." to the routine's body string. \ pop eax  
Slap the routine's routine header on the routine's header string.  
Slap the routine's routine body on the routine's body string.  
Compile the header of the routine.

To add the finalize routine for a type (record):  
If the type is nil, exit.  
If the compiler's abort flag is set, exit.  
Add a routine to the routines given nil.  
Append "~finalize a " then the type's name to the routine's header string.  
Append "intel $50. " to the routine's body string. \ push eax  
Loop.  
Get a field from the type's fields.  
If the field is nil, break.  
If the field's type should not be finalized, repeat.  
Append "~finalize the " then the type's name then "'s " then the field's name then ". " to the routine's body string.  
Repeat.  
Append "intel $58." to the routine's body string. \ pop eax  
Slap the routine's routine header on the routine's header string.  
Slap the routine's routine body on the routine's body string.  
Compile the header of the routine.

Note the tildes ("~") in the routine names; we don't expect the programmer will ever call these routines directly.

152. And these are the deciders we need to help us do that:

To decide if a type should be finalized:  
If the type is nil, say no.  
If the type can be reduced to "substring", say no.  
If the type can be reduced to "string", say yes.  
If the type's fields should be finalized, say yes.  
Say no.

To decide if some fields should be finalized:  
Get a field from the fields.  
If the field is nil, say no.  
If the field's type should be finalized, say yes.  
Repeat.

153. This is how we similarly add the allocate routine for a type:

To add the allocate routine for a type:  
If the type is nil, exit.  
If the compiler's abort flag is set, exit.  
If the type cannot be reduced to "pointer", exit.  
If the type's target type is nil, exit.  
Add a routine to the routines given nil.  
Append "allocate memory for a " then the type's name to the routine's header string.  
Append "assign the " then the type's name then " given " to the routine's body string.  
Append the type's target type's length then "." to the routine's body string.  
Slap the routine's routine header on the routine's header string.  
Slap the routine's routine body on the routine's body string.  
Compile the header of the routine.

Note that there is no tilde in the routine name; we *do* expect the programmer to call these routines directly.

154. This is the routine that adds deallocate routines for dynamic types:

To add the deallocate routine for a type:  
If the type is nil, exit.  
If the compiler's abort flag is set, exit.  
If the type cannot be reduced to "pointer", exit.  
If the type's target type is nil, exit.  
Add a routine to the routines given nil.  
Append "deallocate a " then the type's name to the routine's header string.  
Append "if the " then the type's name then " is nil, exit. " to the routine's body string.  
If the type's target type should be finalized, append "~finalize the " then the type's name then "'s target. " to the routine's body string.  
Append "unassign the " then the type's name then "." to the routine's body string.  
Slap the routine's routine header on the routine's header string.  
Slap the routine's routine body on the routine's body string.  
Compile the header of the routine.

Again, no tilde in the routine name since a programmer may want to call these routines directly.

155. And these are the routines that add the destroy routines for each dynamic type:

To add the destroy routine for a type:  
If the type is nil, exit.  
If the compiler's abort flag is set, exit.  
If the type should not be forgotten, exit.  
If the type can be reduced to "things", add the destroy routine for the type (things); exit.  
If the type can be reduced to "pointer", add the destroy routine for the type (pointer); exit.

To add the destroy routine for a type (things):  
If the type is nil, exit.  
If the compiler's abort flag is set, exit.  
If the type's fields' first is nil, abort with "Internal error - add the forget routine for a type (things)"; exit.  
If the type's fields' first's type is nil, abort with "Internal error 2 - add the forget routine for a type (things)"; exit.  
Put "destroy [" then the type's name then "]" into a moniker.  
If the moniker is in the routine index, exit.  
Add a routine to the routines given nil.  
Append "destroy a " then the type's name to the routine's header string.  
Append "if the " then the type's name then "'s first is nil, exit. " to the routine's body string.  
Put the type's fields' first's type's name into a name.  
Append "put the " then the type's name then "'s first into a " then the name then ". " to the routine's body string.  
Append "remove the " then the name then " from the " then the type's name then ". " to the routine's body string.  
Append "destroy the " then the name then ". " to the routine's body string.  
Append "repeat. " to the routine's body string.  
Slap the routine's routine header on the routine's header string.  
Slap the routine's routine body on the routine's body string.  
Compile the header of the routine.

To add the destroy routine for a type (pointer):  
If the type is nil, exit.  
If the compiler's abort flag is set, exit.  
Put "destroy [" then the type's name then "]" into a moniker.  
If the moniker is in the routine index, exit.  
Add a routine to the routines given nil.  
Append "destroy a " then the type's name to the routine's header string.  
Append "if the " then the type's name then " is nil, exit. " to the routine's body string.  
If the type's target type can be reduced to "pointer", append "destroy the " then the type's name then "'s target. " to the routine's body string.  
If the type's target type can be reduced to "things", append "destroy the " then the type's name then "'s target. " to the routine's body string.  
Loop.  
Get a field from the type's target type's fields.  
If the field is nil, break.  
If the field's reference flag is set, repeat.  
If the field's type should not be forgotten, repeat.  
Append "destroy the " then the type's name then "'s " then the field's name then ". " to the routine's body string.  
Repeat.  
Append "deallocate the " then the type's name then ". " to the routine's body string.  
Slap the routine's routine header on the routine's header string.  
Slap the routine's routine body on the routine's body string.  
Compile the header of the routine.

Note that we expect the programmer to call these routines directly.

Note also that they "go deep" -- if anything attached to a record that is being destroyed needs to be destroyed, these routines let the programmer simply "forget" about those attached things. This spares the programmer a lot of repetative coding.

156. Finally, we need this routine to decide if a type can be forgotten by the programmer:

To decide if a type should be forgotten:  
If the type is nil, say no.  
If the type's name is "thing", say no.  
If the type's name is "things", say no.  
If the type can be reduced to "things", say yes.  
If the type can not be reduced to "pointer", say no.  
If the type's target type is nil, say no.  
If the type's target type cannot be reduced to "pointer", say yes.  
If the type's target type should not be forgotten, say no.  
Say yes.

157. And that's it for this step. Let's run it. Then, in the blue, load up the daughter and list her. Then look in the listing. The first "~finalize" routine in the list is this one...

/routine/~finalize [~inline compiler type]/no/no/no/no//0/0/00000000/  
/variable/parameter/no/~inline compiler type/~inline compiler type/~inline compiler type/~inline compiler type/00000000/no/1/no////

...which is needed because the "~inline compiler type" has lots of strings in it, as you can see from the type definition (earlier in the listing):

/type/~inline compiler type/~inline compiler types/00000040/record/record///0/  
/variable/field/no/directory/directory/directory/directory/00000000/no/1/no////  
/variable/field/no/timer/timer/timer/timer/00000008/no/1/no////  
/variable/field/no/abort flag//flag/flag/00000014/no/1/no////  
/variable/field/no/abort message/abort message/abort message/abort message/00000018/no/1/no////  
/variable/field/no/abort path//path/path/00000020/no/1/no////  
/variable/field/no/abort row#//row#/row#/00000028/no/1/no////  
/variable/field/no/name count/name/count/count/0000002C/no/1/no////  
/variable/field/no/exe buffer/exe/buffer/buffer/00000030/no/1/no////  
/variable/field/no/listing buffer/listing/buffer/buffer/00000038/no/1/no////

Recall that a directory is a path and a path is a string, and that buffer is also a string.

And the first generated "allocate" routine in the listing is this one...

/routine/allocate memory for [entry]/no/no/no/no//0/0/00000000/  
/variable/parameter/no/entry/entry/entry/entry/00000000/no/1/no////

Followed by the associated "deallocate" and "destroy" routines:

/routine/deallocate [entry]/no/no/no/no//0/0/00000000/  
/variable/parameter/no/entry/entry/entry/entry/00000000/no/1/no////

/routine/destroy [entry]/no/no/no/no//0/0/00000000/  
/variable/parameter/no/entry/entry/entry/entry/00000000/no/1/no////  
  
158. Extra credit exercise: You probably noticed that the generated routines above have no meat in them. That's because we don't list routine bodies until later when they've been compiled. But let's take a peek. In the gray, get into the son's compiler, find the "list a routine" routine, and add this line (including the comment at the end)...

If the routine's moniker starts with "~", append the routine's body string to the buffer; advance the buffer. \\*\*\* testing only

...just before the "Advance the buffer" line at the end of that routine.

Now run the son, open and list the daughter, and examine the listing. The finalize routine for our "~inline compiler type" now looks like this (without the word-wrap):

/routine/~finalize [~inline compiler type]/no/no/no/no//0/0/00000000/  
/variable/parameter/no/~inline compiler type/~inline compiler type/~inline compiler type/~inline compiler type/00000000/no/1/no////  
intel $50. ~finalize the ~inline compiler type's directory. ~finalize the ~inline compiler type's abort message. ~finalize the ~inline compiler type's abort path. ~finalize the ~inline compiler type's exe buffer. ~finalize the ~inline compiler type's listing buffer. intel $58.

Those sentences will get compiled later with the rest of the programmer's (and our generated) code. Since directories, abort messages, abort paths, and buffers are all just strings, each of those sentences will eventually resolve into a call to this finalizer...

/routine/~finalize [string]/no/no/no/no//0/0/00000000/  
/variable/parameter/no/string/string/string/string/00000000/no/1/no////  
intel $50. unassign the string's first. intel $58.

...who gives the string's memory back to Windows by calling this Noodle routine:

To unassign a pointer:  
If the pointer is nil, exit.  
Call "kernel32.dll" "HeapFree" with the heap pointer and 0 [no options] and the pointer returning a number.  
If the number is 0, exit.  
Void the pointer.  
Subtract 1 from the heap count.

And that's the way we manage string memory for the programmer, and help him conveniently deal with the rest of his dynamically-allocated data.

159. Now delete that temporary "\\*\*\* testing only" line in the son's compiler.

---

Matching a statement (in the body of a routine) with the routine (that should be called to do the work) is mostly a matter of (a) breaking down the statement into a series of monikettes, and then (b) finding a routine header with the same monikettes. We do this in the next step after this one. In this step, we increase our chances of success by re-indexing some of our routine headers using "reduced type" monikettes. For example, this routine...

To divide a number by a number giving a quotient and a remainder:

...which at this point is indexed as...

divide [number] by [number] giving [quotient] and [remainder]

...will be re-indexed as...

divide [number] by [number] giving [number] and [number]

...since both "quotient" and "remainder" are elsewhere defined as numbers.

So let's get to work.

160. In the gray, open up the CAL's son and adjust the "compile a directory" routine to look like this:

To compile a directory:  
Compile the directory (start).  
Compile the directory (load the source files).  
Compile the directory (scan the source files).  
Compile the directory (resolve the types).  
Compile the directory (resolve the globals).  
Compile the directory (compile the headers of the routines).  
Compile the directory (calculate lengths and offsets of types).  
Compile the directory (add the built-in memory routines).  
Compile the directory (index the routines for utility use).  
Abort with "I'm not old enough to make babies.".  
\Compile the directory (compile the bodies of the routines).  
\Compile the directory (add and compile the built-in startup routine).  
\Compile the directory (offset parameters and variables).  
\Compile the directory (address).  
\Compile the directory (transmogrify).  
\Compile the directory (link).  
\Compile the directory (write the exe).  
\Compile the directory (stop).

161. Then add these three guys back in:

To compile a directory (index the routines for utility use):  
If the compiler's abort flag is set, exit.  
Show status "Utilitizing...".  
Start the indexing utilities timer.  
Index the routines for utility use.  
Scrub the utility index.  
Reindex the utility index.  
Stop the indexing utilities timer.

To index some routines for utility use:  
If the compiler's abort flag is set, exit.  
Get a routine from the routines.  
If the routine is nil, exit.  
Index the routine for utility use.  
Repeat.

To index a routine for utility use:  
If the compiler's abort flag is set, exit.  
If the routine is nil, exit.  
If the routine's function flag is set, exit.  
If the routine's parameters' count is 0, exit.  
Copy the routine's monikettes into some monikettes.  
Reduce the monikettes for utility use.  
Convert the monikettes to a moniker.  
Destroy the monikettes.  
If the moniker is in the routine index, exit.  
Find a refer given the moniker and the utility index.  
If the refer is not nil, put nil into the refer's pointer; exit.  
Index the routine given the moniker and the utility index.

Note that if the reduced moniker is the same as an existing routine's moniker, we don't put it in the Utility Index.

Note also that if two or more routines reduce to the same moniker, we mark the first one as invalid by putting nil into the first routine's refer's pointer (in the Utility Index), so we'll know to remove it later. We don't want to remove it now, because then we wouldn't catch any other routines that ambiguously reduce to the same moniker.

162. So far, so good. Now we need the helper routines, starting with these two:

To copy some monikettes into some other monikettes:  
Destroy the other monikettes.  
Loop.  
Get a monikette from the monikettes.  
If the monikette is nil, exit.  
Copy the monikette into another monikette.  
Append the other monikette to the other monikettes.  
Repeat.

To copy a monikette into another monikette:   
If the monikette is nil, void the other monikette; exit.  
Allocate memory for the other monikette.  
Put the monikette's string into the other monikette's string.  
Put the monikette's type into the other monikette's type.  
Put the monikette's variable into the other monikette's variable.  
\ don't copy substring  
Put the monikette's current type into the other monikette's current type.

163. And this little looper...

To reduce some monikettes for utility use:  
Get a monikette from the monikettes.  
If the monikette is nil, exit.  
If the monikette's type is nil, repeat.  
Reduce the monikette's type to a type for utility use.  
If the type is not nil, put the type into the monikette's type.  
Repeat.

164. ...who calls the routine below, who keeps reducing the type's type until he hits a fundamental type (or can't go any further):

To reduce a type to another type for utility use:  
Put the type into the other type.  
Loop.  
If the other type is nil, exit.  
If the other type's name is "hex string", exit.  
If the other type's name is "string", exit.  
If the other type's name is "number", exit.  
If the other type's name is "pointer", exit.  
If the other type's name is "thing", void the other type; exit.  
If the other type's base type is the other type, void the other type; exit.  
Put the other type's base type into the other type.  
Repeat.

165. These are the routines we use to get those ambiguous monikers out of the Utility Index:

To scrub an index:  
Get a bucket given the index.  
If the bucket is nil, exit.  
If the bucket's refers are empty, repeat.  
Scrub the bucket's refers.  
Repeat.

To scrub some refers:  
Swap the refers with some other refers.  
Loop.  
Put the other refers' first into a refer.  
If the refer is nil, exit.  
Remove the refer from the other refers.  
If the refer's pointer is nil, destroy the refer; repeat.  
Append the refer to the refers.  
Repeat.

166. And these are the routines we use to add an additional Routine Index entry for each routine that unambiguously reduced to something more fundamental:

To reindex the utility index:  
If the compiler's abort flag is set, exit.  
Get a bucket given the utility index.  
If the bucket is nil, exit.  
If the bucket's refers are empty, repeat.  
Reindex the utility index given the bucket's refers.  
Repeat.

To reindex the utility index given some refers:  
Get a refer from the refers.  
If the refer is nil, exit.  
Put the refer's pointer into a routine.  
If the routine is nil, abort with "Internal error - index the untility routines given some refers"; exit.  
Copy the routine's monikettes into some monikettes.  
Reduce the monikettes for utility use.  
Index the routine given the monikettes and the routine index.  
Destroy the monikettes.  
Repeat.

The Utility Index is not used again, since the Routine Index now includes both the original and the reduced header (if any) for each routine. We keep the Utility Index around, however, as a debugging aid.

167. Run it. In the blue, open up the son's daughter and list her. The open up her listing file and find "utility index:" (with the colon, but without the quotes). At the top of that section you should see something like this:

UTILITY INDEX:

/658 buckets/717 refers/

/bucket/  
/refer/divide [number] by [number] giving [number] and [number]/

168. And at the bottom we can see that all this reducing and scrubbing and reindexing doesn't take much time at all:

TIMERS:

/timer/loading/125/  
/timer/scanning/469/  
/timer/resolving types/31/  
/timer/resolving globals/110/  
/timer/compiling routine headers/281/  
/timer/calculating/0/  
/timer/adding built-in memory routines/62/  
/timer/indexing utilities/78/  
/timer/compiling routine bodies/0/  
/timer/adding built-in startup routine/0/  
/timer/offsetting/0/  
/timer/addressing/0/  
/timer/transmogrifying/0/  
/timer/linking/0/  
/timer/writing/0/  
/timer/total/1641/

See you next time!

---

This is the big one. Let's take a moment to review.

At this point, the compiler has cataloged the *vocabulary* and *grammar* that the programmer wants to use (as defined by the *types*, the *global variables*, the *literals*, and the *routine headers* that the Compiler found in his source). For quick and easy access, the Type Index has an entry pointing to each *type* definition, the Global Index has an entry pointing to each *global variable* definition, and the Literal Index has an entry pointing to the definition of each *literal*. The Routine Index is a little more robust, and has multiple entries for each *routine header,* one monikette at a time, from left-to-right. For example, this routine...

To add a number to another number:

...is cataloged in the Routine Index like so:

add...

add [number]...

add [number] in/into/to....

add [number] in/into/to [number]

This arrangement makes it easier to match each sentence in the body of each routine with a corresponding routine. Once we know who to call, all we have to do is make a list of the logical machine-code *fragments* that will be needed to push the correct parameters and actually make the call. Each routine we call will have a body of it's own, of course, and may, in turn, call other routines, etc, etc. At the very bottom, however, we will find routines that have bodies containing only machine code, like this one...

To bump a rider:  
Intel $8B9D08000000. \ mov ebx,[ebp+8] \ the rider  
Intel $FF8314000000. \ inc [ebx+20] \ the rider's token's last  
Intel $FF8308000000. \ inc [ebx+8] \ the rider's source's first

...or a call to a Windows function, like this one:

To buzz:   
Call "kernel32.dll" "Beep" with 220 and 200.

Simple enough. Unfortunately, the whole affair is complicated by sentences that require additional processing and/or more than one call to properly execute. This sentence, for example...

Put the box's height divided by 2 into a center offset.

...requires fragments for on-the-fly allocation of a *scratch* variable (to calculate and hold the address of the "height" field in the box record), with more fragments for on-the-fly allocation of an *intermediate* variable (to hold the result we get when we secretly call the Noodle's "divide a number by another number" routine to divide the box's height by 2), with even more fragments for on-the-fly allocation of the *local* "center offset" target variable that will be passed to the Noodle's bottom-level "put a number into another number" routine to fill in.

So let's get to it.

169. In the gray, move our abort line down and uncomment the next step:

To compile a directory:  
Compile the directory (start).  
Compile the directory (load the source files).  
Compile the directory (scan the source files).  
Compile the directory (resolve the types).  
Compile the directory (resolve the globals).  
Compile the directory (compile the headers of the routines).  
Compile the directory (calculate lengths and offsets of types).  
Compile the directory (add the built-in memory routines).  
Compile the directory (index the routines for utility use).  
Compile the directory (compile the bodies of the routines).  
Abort with "I'm not old enough to make babies.".  
\Compile the directory (add and compile the built-in startup routine).  
\Compile the directory (offset parameters and variables).  
\Compile the directory (address).  
\Compile the directory (transmogrify).  
\Compile the directory (link).  
\Compile the directory (write the exe).  
\Compile the directory (stop).

170. Then add in these three guys:

To compile a directory (compile the bodies of the routines):  
If the compiler's abort flag is set, exit.  
Show status "Compiling...".  
Start the compiling routine bodies timer.  
Compile the bodies of the routines.  
Stop the compiling routine bodies timer.

To compile the bodies of the routines:  
Compile the body of the routine called "initialize before run".  
Compile the body of the routine called "run".  
Compile the body of the routine called "finalize after run".

To compile the body of the routine called a string:  
If the compiler's abort flag is set, exit.  
Find a routine given the string and the routine index.  
If the routine is nil, abort with "I need a routine to '" then the string then "'."; exit.  
Compile the body of the routine.

Note that we're not just looping thorough the routines in the order we found them; rather, we're starting with the Noodle's "initialize before run" routine (and anyone he calls), then we move on to the programmer's "run" routine (and whomever he calls), and lastly we compile the Noodle's "finalize after run" routine (and his helpers, if any). This is how we avoid compiling the bodies of routines that never get called.

171. This is the top-level routine-body compiler:

To compile the body of a routine:  
If the routine is nil, exit.  
If the compiler's abort flag is set, exit.  
If the routine's compiled flag is set, exit.  
Set the routine's compiled flag.  
Slap a rider on the routine's routine body.  
Move the rider (compiler rules).  
If the rider's token is the semi-colon byte, compile the body of the routine given the rider (alternate wording); exit.

If the rider's token is "employ", compile the body of the routine given the rider (employ); exit.  
Put the current routine into a saved routine.  
Put the routine into the current routine.  
Create the routine's nickname index given 101.  
Eliminate duplicate nicknames from the routine's parameters given the routine's nickname index.  
Compile the body of the routine (prolog).  
Loop.  
If the compiler's abort flag is set, exit.  
If the rider's token is blank, break.  
Compile the next statement given the rider.  
Repeat.  
Compile the body of the routine (epilog).  
Put the saved routine into the current routine.

After dealing with possible special cases ("alternate wordings" and "employs"), this guy creates a nickname index for the routine's parameters, and then compiles the routine's *prolog*, *body proper*, and *epilog*.

172. This is the guy we need to handle "alternate wordings" (implicit replacement routines):

To compile the body of a routine given a rider (alternate wording):

If the compiler's abort flag is set, exit.  
Move the rider (compiler rules).

Put the routine's next into a target routine.  
If the target routine is nil, abort with "You've defined an alternate wording here, but I'm not sure what you're rewording." and the rider; exit.  
Put the target routine's moniker into the routine's employs moniker.  
Compile the body of the target routine.

173. And this is the helper we need to handle "employs" (explicit replacement routines):

To compile the body of a routine given a rider (employ):  
If the compiler's abort flag is set, exit.  
Move the rider (compiler rules).  
Compile a routine reference given the rider.  
Convert the routine reference to the routine's employs moniker.  
Destroy the routine reference.  
If the rider's token is not ".", abort with "Employ statments should end with a period not '" then the rider's token then "'." and the rider; exit.  
Move the rider (compiler rules).  
If the rider's token is not blank, abort with "Once an employ, only an employ" and the rider; exit.  
Find another routine given the routine's employs moniker and the routine index.  
If the other routine is nil, abort with "I can't find the '" then the routine's employs moniker then "' routine you're trying to employ." and the routine's locus; exit.  
Compile the body of the other routine.

174. We're going to need some additional types, mostly to deal with mathematical expressions, from here on. May as well put them all in now:

A phrase is a string.

A term has a variable and a phrase.

A field term has  
a dereference flag,  
a field (reference),  
a function routine (reference),  
a push flag.

An expression is a term.

An intermediate is a local.

A scratch is a local.

A nibble substring is a substring.

A prototype string has a first byte pointer and a last byte pointer.

A *term* is either a *literal*, a *new local variable*, or a reference to a *previously defined variable*. A numeric term may have a minus or plus sign in front of it. We call these "negated" and "posigated" terms, respectively. A record term may be followed by any number of possessives indicating fields within the record (like "box's left" or "rider's token's first").

An *expression* is two or more terms separated by the infix operators *plus*, *minus*, *times*, *divided by* and *then*. We say that "an expression is a term" because the result of an expression (after execution) is really just another term.

175. This is the top-level expression compiler:

To compile an expression given a rider:  
If the compiler's abort flag is set, exit.  
Compile a term given the rider.  
Put the term into the expression.  
Loop.  
If the compiler's abort flag is set, exit.  
If the rider's token is not any operator, exit.  
If the rider's token is "divided", compile the expression given the rider (divided); repeat.  
If the rider's token is "minus", compile the expression given the rider (minus); repeat.  
If the rider's token is "plus", compile the expression given the rider (plus); repeat.  
If the rider's token is "then", compile the expression given the rider (then); repeat.  
If the rider's token is "times", compile the expression given the rider (times); repeat.

176. This is the top-level term compiler, together with one of his trivial helpers:

To compile a term given a rider:  
Clear the term.  
If the compiler's abort flag is set, exit.  
Compile the term given the rider (common part).  
If the rider is on any simile, exit.

If the rider's token is "AS", compile the term given the rider (as part).  
If the rider's token is "/", compile the term given the rider (runtime ratio).

To clear a term:  
Void the term's variable.  
Clear the term's phrase.

177. And this is the guy who compiles the part that all terms have:

To compile a term given a rider (common part):  
If the compiler's abort flag is set, exit.  
If the rider's token is the dash byte, compile the term given the rider (negated term); exit.  
If the rider's token is the cross byte, compile the term given the rider (posigated term); exit.  
If the rider's token is the start of any variable, compile the term given the rider (variable); exit.  
If the rider's token is the start of any new local, compile the term given the rider (new local); exit.  
If the rider's token is any literal, compile the term given the rider (literal); exit.  
Abort with "I expected a term hereabouts but all I found was '" then the rider's token then "'." and the rider.

178. This guy deals with terms that have a minus sign in front of them:

To compile a term given a rider (negated term):  
If the compiler's abort flag is set, exit.  
Move the rider (compiler rules).  
Compile another term given the rider.  
If the compiler's abort flag is set, exit.  
If the other term's variable is nil, abort with "Invalid negate, I don't understand the variable '" then the other term's phrase then "'." and the rider; exit.  
Add an intermediate given the other term's type name and the rider's token's first.  
Add several fragments given "put" and the other term's variable and "into" and the intermediate and the other term's locus.  
Add two fragments given "negate" and the intermediate and the rider's token's first.  
Put the intermediate into the term's variable.  
Clear the term's phrase.

179. These three routines are general-purpose helpers:

To add an intermediate given a type name and a locus:  
If the current routine is nil, void the intermediate; exit.  
Add the intermediate to the current routine's locals given "local" and the locus.  
Generate the intermediate's name given "~I".  
Put the type name into the intermediate's type name.  
Resolve the intermediate.

To add several fragments given a string and a variable and another string and another variable and a locus:  
Add a fragment given the push address tag and the other variable.  
Add another fragment given the push address tag and the variable.  
Add a monikette to some monikettes given the string.  
Add another monikette to the monikettes given the variable's type.  
Add a third monikette to the monikettes given the other string.  
Add a fourth monikette to the monikettes given the other variable's type.  
Find a routine given the monikettes.  
If the routine is nil, convert the monikettes to a moniker.  
Destroy the monikettes.  
If the routine is nil, abort with "I don't know how to '" then the moniker then "'." and the locus; exit.  
Add a third fragment given the call internal tag and the routine.

To add a fragment given a tag and a variable:  
If the current routine is nil, void the fragment; exit.  
Create the fragment given the tag.  
Append the fragment to the current routine's fragments.  
Put the variable into the fragment's variable.  
Set the compiled flag in the variable.

180. And this is a special helper:

To set the compiled flag in a variable:  
If the variable is nil, exit.  
If the variable's compiled flag is set, exit.  
Set the variable's compiled flag.  
If the variable's kind is not "global", exit.  
If the variable's literal is nil, exit.  
Find a routine given "put" and the variable's literal's type and "into" and the variable's type.  
If the routine is nil, find the routine given "convert" and the variable's literal's type and "to" and the variable's type; set a flag.  
If the routine is nil, abort with "The type of this global and the type of its literal are not compatible." and the variable's locus; exit.  
Compile the body of the routine.  
If the flag is set, put "convert the " then the variable's literal's name then " to the " then the variable's name then ". " into the variable's initializer string.  
If the flag is not set, put "put the " then the variable's literal's name then " into the " then the variable's name then ". " into the variable's initializer string.

If the variable is a global, and the programmer wants us to initialize it with a literal value, we're going to need a "put" or "convert" routine to actually fill it in at run time, so we make sure we've got such a routine now. If we find one, we take a moment to compile its body, and then we fill in the variable's "initializer string" with the Plain English sentence that will do the initializing. "Global initializer" sentences are compiled in the next step.

181. Here are the seven helpers we need to find the right routines to call:

To find a routine given a string and a type and another string and another type:  
Add a monikette to some monikettes given the string.  
Add another monikette to the monikettes given the type.  
Add a third monikette to the monikettes given the other string.  
Add a fourth monikette to the monikettes given the other type.  
Find the routine given the monikettes.  
Destroy the monikettes.

To find a routine given some monikettes:  
Void the routine.  
Find the routine given a moniker and the monikettes' first and the routine index.

To find a routine given a moniker and a monikette and an index:  
If the monikette is nil, exit.  
Void the routine.  
Privatize the moniker.  
Put the monikette's type into the monikette's current type.  
Slap the monikette's current substring on the monikette's string.  
Loop.  
Append the monikette to the moniker (while bubbling).  
Find a refer given the moniker and the index.  
If the refer is not nil, put the refer's pointer into the routine.  
If the refer is not nil, find the routine given the moniker and the monikette's next and the index.  
If the routine is not nil, break.  
Bubble the monikette.  
If the monikette is bubbled out, break.  
Put the original moniker into the moniker.  
Repeat.

To append a monikette to a moniker (while bubbling):  
If the monikette is nil, exit.  
If the moniker is not blank, append the space byte to the moniker.  
If the monikette's current substring is not blank, append the monikette's current substring to the moniker; exit.  
If the monikette's current type is not nil, append "[" and the monikette's current type's name and "]" to the moniker (fast); exit.

To bubble a monikette:  
If the monikette is nil, exit.  
If the monikette's current substring is not blank, clear the monikette's current substring; exit.  
If the monikette's current type is nil, exit.  
If the monikette's current type is any built-in type, put nil into the monikette's current type; exit.  
Put the monikette's current type's base type into the monikette's current type.

To decide if a type is any built-in type:  
If the type is nil, say no.  
If the type is the type's base type, say yes.  
Say no.

To decide if a monikette is bubbled out:  
If the monikette is nil, say yes.  
If the monikette's current type is not nil, say no.  
If the monikette's current substring is not blank, say no.  
Say yes.

182. And here are two more routines we need to help us manage our fragments:

To create a fragment given a tag:  
Allocate memory for the fragment.  
Put the tag into the fragment's tag.

To add a fragment given a tag and a routine:  
If the current routine is nil, void the fragment; exit.  
Create the fragment given the tag.  
Append the fragment to the current routine's fragments.  
Put the routine into the fragment's routine.  
Compile the body of the routine.

To add two fragments given a string and a variable and a locus:  
Add a fragment given the push address tag and the variable.  
Add a monikette to some monikettes given the string.  
Add another monikette to the monikettes given the variable's type.  
Find a routine given the monikettes.  
If the routine is nil, convert the monikettes to a moniker.  
Destroy the monikettes.  
If the routine is nil, abort with "I don't know how to '" then the moniker then "'." and the locus; exit.  
Add another fragment given the call internal tag and the routine.

183. And now we're ready to compile "posigated" terms:

To compile a term given a rider (posigated term):  
If the compiler's abort flag is set, exit.  
Move the rider (compiler rules).  
Compile the term given the rider.  
If the term's variable is nil, abort with "Invalid posigate, I don't understand the variable '" then the term's phrase then "'." and the rider; exit.  
Clear the term's phrase.

184. And "variable" terms, including those followed by possessives:

To decide if a string is the start of any variable:  
If the string is "the", say yes.  
Say no.

To compile a term given a rider (variable):  
If the compiler's abort flag is set, exit.  
If the current routine is nil, abort with "I don't know what you're trying to pull here." and the rider; exit.  
Move the rider (compiler rules).  
Scan a name given the rider.  
Find a variable given the name.  
Put the variable into the term's variable.  
Put "the " into the term's phrase.  
Append the name to the term's phrase.  
Compile the term given the rider (possessives).

To compile a term given a rider (possessives):  
If the compiler's abort flag is set, exit.  
If the rider's token is not any possessive, exit.  
Compile the term given the rider (possessive).  
Repeat.

To compile a term given a rider (possessive):  
If the compiler's abort flag is set, exit.  
Move the rider (compiler rules).  
Scan a name given the rider.  
If the compiler's abort flag is set, exit.  
If the name is "magnitude", compile the term given the rider (possessive - magnitude); exit.  
If the name is "target", compile the term given the rider (possessive - target); exit.  
If the name is "whereabouts", compile the term given the rider (possessive - whereabouts); exit.  
Compile the term given the rider and the name (possessive - field).

185. Some possessives reference compiler-defined metadata. A term's *magnitude*, for example, is it's length in bytes; a term's *target* is the type of data a pointer points to; and a term's *whereabouts* is its address in memory.

To compile a term given a rider (possessive - magnitude):  
If the compiler's abort flag is set, exit.  
If the term's variable is nil, abort with "Invalid use of magnitude, I don't understand the variable '" then the term's phrase then "'." and the rider; exit.  
Add a literal to the literals given the rider's token's first.  
Convert the term's type's length to the literal's data.  
Put "number" into the literal's type name.  
Resolve the literal.  
Put the literal into the term's variable.  
Clear the term's phrase.

To compile a term given a rider (possessive - target):  
If the compiler's abort flag is set, exit.  
If the term's variable is nil, abort with "Invalid use of target, I don't understand the variable '" then the term's phrase then "'." and the rider; exit.  
If the term's type cannot be reduced to "pointer", abort with "You can only get the target of pointer types." and the rider; exit.  
Compile the term given the rider (dereference).

To compile a term given a rider (possessive - whereabouts):  
If the compiler's abort flag is set, exit.  
If the term's variable is nil, abort with "Invalid use of whereabouts, I don't understand the variable '" then the term's phrase then "'." and the rider; exit.  
Add an intermediate given "pointer" and the rider's token's first.   
Add a fragment given the load address tag and the term's variable and the intermediate.  
Put the intermediate into the term's variable.  
Clear the term's phrase.

186. Some terms need to be "dereferenced" to be used. If, for example, we're passed the address of the address of something, we need to get past that first address to get to the thing we're actually interested in.

To compile a term given a rider (dereference):  
If the compiler's abort flag is set, exit.  
If the term's variable is nil, abort with "Invalid dereference, I don't understand the variable '" then the term's phrase then "'." and the rider; exit.  
If the term's type's target type is nil, abort with "I'm unable to determine the type of this target." and the rider; exit.  
If the term's kind is "scratch", compile the term given the rider (dereference - in place); exit.  
Add a scratch given the term's type's target type's name and the rider's token's first.  
Add a fragment given the load address tag and the term's variable and the scratch.  
Add another fragment given the dereference tag and the scratch.  
Put the scratch into the term's variable.  
Clear the term's phrase.

To compile a term given a rider (dereference - in place):  
If the term's variable is nil, abort with "Internal error - compile a term given a rider (dereference - in place)" and the rider; exit.  
Put the term's type's target type's name into the term's type name.  
Put the term's type's target type into the term's type.  
Add a fragment given the dereference tag and the term's variable.  
Clear the term's phrase.

187. Here are a couple more general-purpose helpers we need:

To add a scratch given a type name and a locus:  
If the current routine is nil, void the scratch; exit.  
Add the scratch to the current routine's locals given "scratch" and the locus.  
Generate the scratch's name given "~S".  
Put the type name into the scratch's type name.  
Resolve the scratch.

To add a fragment given a tag and a variable and another variable:  
If the current routine is nil, void the fragment; exit.  
Create the fragment given the tag.  
Append the fragment to the current routine's fragments.  
Put the variable into the fragment's variable.  
Put the other variable into the fragment's other variable.  
Set the compiled flag in the variable.  
Set the compiled flag in the other variable.

188. And these are the four guys we need to handle references to fields in records:

To compile a term given a rider and a name (possessive - field):  
If the compiler's abort flag is set, exit.  
If the term's variable is not nil, put the term's type into a type.  
Find a field term given the term's phrase and the type and the name.  
If the field term is not empty, compile the term given the rider and the field term; exit.  
If the term's variable is nil, abort with "I don't understand the variable '" then the term's phrase then "'." and the rider; exit.  
If the type can be reduced to "pointer", put the type's target type into the type.  
If the type is nil, abort with "There's no '" then the name then "' field in a '" then the term's type's name then "'." and the rider; exit.  
Loop.  
If the compiler's abort flag is set, exit.  
Get a field from the type's fields.  
If the field is nil, abort with "There's no '" then the name then "' field in a '" then the term's type's name then "'." and the rider; exit.  
Find a deep field term given "" and the field's type and the name.  
If the deep field term is empty, repeat.  
Put the field into a first field term's field.  
If the term's type can be reduced to "pointer", set the first field term's dereference flag.  
Compile the term given the rider and the first field term.  
Compile the term given the rider and the deep field term.

To find a field term given a phrase and a type and a name:  
Clear the field term.  
Find the field term given the type and the name.  
If the field term's field is not nil, exit.  
If the phrase is not blank, find a routine given "put" and the phrase and "'s " then the name and "into".  
If the routine is not nil, put the routine into the field term's function; exit.  
If the type is nil, exit.  
Find the routine given "put" and the type and "'s " then the name and "into".  
If the routine is not nil, set the field term's push flag; put the routine into the field term's function; exit.

Make sure you leave the last three double-commented lines of the above routine (not shown here) out.

To find a field term given a type and a name:  
If the type is nil, exit.  
Find a field given the name and the type's fields.  
If the field is not nil, put the field into the field term's field; exit.  
If the type's target type is not nil, find the field given the name and the type's target type's fields.  
If the field is not nil, set the field term's dereference flag; put the field into the field term's field; exit.

To clear a field term:  
Fill bytes with the null byte starting at the field term's whereabouts for the field term's magnitude.

189. These are just another three general-purpose helpers we need:

To find a routine given a string and another string and a third string and a fourth string:  
Add a monikette to some monikettes given the string.  
Add another monikette to the monikettes given the other string.  
Add a third monikette to the monikettes given the third string.  
Add a fourth monikette to the monikettes given the fourth string.  
Find the routine given the monikettes.  
Destroy the monikettes.

To find a routine given a string and a type and another string and a third string:  
Add a monikette to some monikettes given the string.  
Add another monikette to the monikettes given the type.  
Add a third monikette to the monikettes given the other string.  
Add a fourth monikette to the monikettes given the third string.  
Find the routine given the monikettes.  
Destroy the monikettes.

To find a routine given a string and a type and another string and a third string and a fourth string and a fifth string:  
Add a monikette to some monikettes given the string.  
Add another monikette to the monikettes given the type.  
Add a third monikette to the monikettes given the other string.  
Add a fourth monikette to the monikettes given the third string.  
Add a fifth monikette to the monikettes given the fourth string.  
Add a sixth monikette to the monikettes given the fifth string.  
Find the routine given the monikettes.  
Destroy the monikettes.

190. And these are six additional helpers we need to compile fields within records:

To decide if a field term is empty:  
If the field term's field is not nil, say no.  
If the field term's function is not nil, say no.  
Say yes.

To compile a term given a rider and a field term:  
If the compiler's abort flag is set, exit.  
If the field term's dereference flag is set, compile the term given the rider (dereference).    
If the field term's function is not nil, compile the term given the rider and the field term (function); exit.  
If the term's kind is "scratch", compile the term given the rider and the field term (in place); exit.  
Put the field term's field into a field.  
Add a scratch given the field's type's name and the rider's token's first.  
Add a fragment given the load address tag and the term's variable and the scratch.  
If the field's offset is not 0, add a third fragment given the increment tag and the scratch and the field's offset.  
Put the scratch into the term's variable.  
Clear the term's phrase.

To compile a term given a rider and a field term (function):  
If the compiler's abort flag is set, exit.  
Put the field term's function into a routine.  
Put the routine's parameters' last into a parameter.  
If the parameter is nil, abort with "Internal error - compile a term given a rider and an field term (function)." and the rider; exit.  
If the parameter's type is nil, abort with "Internal error 2 - compile a term given a rider and an field term (function)." and the rider; exit.  
Add an intermediate given the parameter's type's name and the rider's token's first.  
Add a fragment given the push address tag and the intermediate.  
If the field term's push flag is set, add another fragment given the push address tag and the term's variable.  
Add a third fragment given the call internal tag and the routine.  
Put the intermediate into the term's variable.  
Clear the term's phrase.

To compile a term given a rider and a field term (in place):  
If the compiler's abort flag is set, exit.  
Put the field term's field into a field.  
Put the field's type's name into the term's type name.  
Put the field's type into the term's type.  
If the field's offset is not 0, add a third fragment given the increment tag and the term's variable and the field's offset.  
Clear the term's phrase.

To add a fragment given a tag and a variable and a number:  
If the current routine is nil, void the fragment; exit.  
Create the fragment given the tag.  
Append the fragment to the current routine's fragments.  
Put the variable into the fragment's variable.  
Put the number into the fragment's number.  
Set the compiled flag in the variable.

To decide if a string is the start of any new local:  
If the string is any indefinite article, say yes.  
Say no.

191. This is the guy who deals with new local variables:

To compile a term given a rider (new local):  
If the compiler's abort flag is set, exit.  
If the current routine is nil, abort with "I don't know what you're trying to pull here." and the rider; exit.  
Add a local to the current routine's locals given "local" and the rider's token's first.  
If the rider's token is "another", put "other" into the local's name.  
Move the rider (compiler rules).  
Scan a name given the rider.  
Extend the local's name with the name.  
If the rider's token is "called", compile the call'd part given the rider and the local.  
Resolve the local.  
If the local is duplicated in the current routine's parameters, abort with "'" then the local's name then "' is already in use as a parameter." and the local's locus; exit.  
If the local is duplicated in the current routine's locals,abort with "'" then the local's name then "' is already in use as a local variable." and the local's locus; exit.  
Eliminate duplicate nicknames given the local and the current routine's nickname index.  
Put the local into the term's variable.  
Clear the term's phrase.  
Compile the term given the rider (possessives).

192. And this is the guy who deals with literal terms:

To compile a term given a rider (literal):  
If the compiler's abort flag is set, exit.  
Compile a literal given the rider.  
Put the literal into the term's variable.  
Clear the term's phrase.  
Compile the term given the rider (possessives).

Note that literals can have possessives; for example...

"abcde"'s magnitude.

...which is 5 (the length of the string).

193. This is a little guy who helps us deal with similes:

To decide if a rider is on any simile:

Privatize the rider.  
If the rider's token is "LIKE", say yes.  
If the rider's token is not "AS", say no.  
Move the rider (compiler rules).  
If the rider's token is any indefinite article, say no.  
Say yes.

194. And this is the guy who deals with "type overrides" (as in "put 0 as a pointer into the other pointer", which coerces the number zero to a pointer type literal):

To compile a term given a rider (as part):  
If the compiler's abort flag is set, exit.  
If the term's variable is nil, abort with "Invalid 'as', I don't understand the variable '" then the term's phrase then "'." and the rider; exit.  
Move the rider (compiler rules).  
If the rider's token is not any indefinite article, abort with "I was hoping for an indefinite article after AS, but I found '" then the rider's token then "'." and the rider; exit.  
Move the rider (compiler rules).  
Scan a type name given the rider.  
Find a type given the type name and the type index.  
If the type is nil, abort with "I can't find the '" then the type name then "' type." and the rider; exit.  
Add a scratch given the type's name and the rider's token's first.  
Add a fragment given the load address tag and the term's variable and the scratch.  
Put the scratch into the term's variable.  
Clear the term's phrase.

195. Most ratios in Plain English programs are defined as literals at compile-time (like "1/2" or "1/3"). But sometimes a programmer might define a ratio at run-time, using other variables (like "Put the percent / 100 into a fraction.") Keep in mind that our Compiler agrees with the great mathematician Leopold Kronecker that  "The dear God created the whole numbers; all else is menschenwerk." So when somebody says something like 0.333333333..., we think, "What he's really trying to say is simply 1/3, 1 part out of 3." In any case, this is the guy who compiles "run-time ratios":

To compile a term given a rider (runtime ratio):  
If the compiler's abort flag is set, exit.  
If the term's variable is nil, abort with "Invalid use of /, I don't understand the variable '" then the term's phrase then "'." and the rider; exit.  
Move the rider (compiler rules).  
Compile another term given the rider.  
If the compiler's abort flag is set, exit.  
If the other term's variable is nil, abort with "Invalid use of /, I don't understand the variable '" then the other term's phrase then "'." and the rider; exit.  
Add an intermediate given "ratio" and the rider's token's first.  
Add several fragments given "put" and the term's variable and "and" and the other term's variable and "into" and the intermediate and the rider's token's first.  
Put the intermediate into the term's variable.  
Clear the term's phrase.

196. And this is another handy-dandy helper we need:

To add several fragments given a string and a variable and another string and another variable and a third string and a third variable and a byte pointer:  
Add a fragment given the push address tag and the third variable.  
Add another fragment given the push address tag and the other variable.  
Add a third fragment given the push address tag and the variable.  
Add a monikette to some monikettes given the string.  
Add another monikette to the monikettes given the variable's type.  
Add a third monikette to the monikettes given the other string.  
Add a fourth monikette to the monikettes given the other variable's type.  
Add a fifth monikette to the monikettes given the third string.  
Add a sixth monikette to the monikettes given the third variable's type.  
Find a routine given the monikettes.  
If the routine is nil, convert the monikettes to a moniker.  
Destroy the monikettes.  
If the routine is nil, abort with "I don't know how to '" then the moniker then "'." and the byte pointer; exit.  
Add a fourth fragment given the call internal tag and the routine.

To put a term into another term:  
Put the term's variable into the other term's variable.  
Put the term's phrase into the other term's phrase.

197. And these are the six guys who handle infix expressions (subordinate clauses) in Plain English sentences:

To compile an expression given a rider (divided):  
If the compiler's abort flag is set, exit.  
Move the rider (compiler rules).  
If the rider's token is not "by", abort with "I really prefer the phrase 'divided BY'." and the rider; exit.  
If the expression's variable is nil, abort with "Invalid 'divided by', I don't understand the variable '" then the expression's phrase then "'." and the rider; exit.  
Move the rider (compiler rules).  
Compile a term given the rider.  
If the compiler's abort flag is set, exit.  
If the term's variable is nil, abort with "Invalid 'divided by', I don't understand the variable '" then the term's phrase then "'." and the rider; exit.  
Add an intermediate given the expression's type name and the rider's token's first.  
Add several fragments given "put" and the expression's variable and "into" and the intermediate and the rider's token's first.  
Add several fragments given "divide" and the intermediate and "by" and the term's variable and the rider's token's first.  
Put the intermediate into the expression's variable.  
Clear the expression's phrase.

To compile an expression given a rider (minus):  
If the compiler's abort flag is set, exit.  
If the expression's variable is nil, abort with "Invalid 'minus', I don't understand the variable '" then the expression's phrase then "'." and the rider; exit.  
Move the rider (compiler rules).  
Compile a term given the rider.  
If the compiler's abort flag is set, exit.  
If the term's variable is nil, abort with "Invalid 'minus', I don't understand the variable '" then the term's phrase then "'." and the rider; exit.  
Add an intermediate given the expression's type name and the rider's token's first.  
Add several fragments given "put" and the expression's variable and "into" and the intermediate and the rider's token's first.  
Add several fragments given "subtract" and the term's variable and "from" and the intermediate and the rider's token's first.  
Put the intermediate into the expression's variable.  
Clear the expression's phrase.

To compile an expression given a rider (plus):  
If the compiler's abort flag is set, exit.  
If the expression's variable is nil, abort with "Invalid 'plus', I don't understand the variable '" then the expression's phrase then "'." and the rider; exit.  
Move the rider (compiler rules).  
Compile a term given the rider.  
If the compiler's abort flag is set, exit.  
If the term's variable is nil, abort with "Invalid 'plus', I don't understand the variable '" then the term's phrase then "'." and the rider; exit.  
Add an intermediate given the expression's type name and the rider's token's first.  
Add several fragments given "put" and the expression's variable and "into" and the intermediate and the rider's token's first.  
Add several fragments given "add" and the term's variable and "to" and the intermediate and the rider's token's first.  
Put the intermediate into the expression's variable.  
Clear the expression's phrase.

To compile an expression given a rider (then):  
If the compiler's abort flag is set, exit.  
If the expression's variable is nil, abort with "Invalid 'then', I don't understand the variable '" then the expression's phrase then "'." and the rider; exit.  
Move the rider (compiler rules).  
Compile a term given the rider.  
If the compiler's abort flag is set, exit.  
If the term's variable is nil, abort with "Invalid 'then', I don't understand the variable '" then the term's phrase then "'." and the rider; exit.  
Add an intermediate given "string" and the rider's token's first.  
Add the put or convert fragments given the expression's variable and the intermediate and the rider's token's first.  
Convert the term for concatenation given the rider.  
Add several fragments given "append" and the term's variable and "to" and the intermediate and the rider's token's first.  
Put the intermediate into the expression's variable.  
Clear the expression's phrase.

To compile an expression given a rider (times):  
If the compiler's abort flag is set, exit.  
If the expression's variable is nil, abort with "Invalid 'times', I don't understand the variable '" then the expression's phrase then "'." and the rider; exit.  
Move the rider (compiler rules).  
Compile a term given the rider.  
If the compiler's abort flag is set, exit.  
If the term's variable is nil, abort with "Invalid 'times', I don't understand the variable '" then the term's phrase then "'." and the rider; exit.  
Add an intermediate given the expression's type name and the rider's token's first.  
Add several fragments given "put" and the expression's variable and "into" and the intermediate and the rider's token's first.  
Add several fragments given "multiply" and the intermediate and "by" and the term's variable and the rider's token's first.  
Put the intermediate into the expression's variable.  
Clear the expression's phrase.

To add the put or convert fragments given a variable and another variable and a locus:  
Add a fragment given the push address tag and the other variable.  
Add another fragment given the push address tag and the variable.  
Find a routine given "put" and the variable's type and "into" and the other variable's type.  
If the routine is nil, find the routine given "convert" and the variable's type and "to" and the other variable's type.  
If the routine is not nil, add a third fragment given the call internal tag and the routine; exit.  
\ error message  
Add a monikette to some monikettes given "put/convert".  
Add another monikette to the monikettes given the variable's type.  
Add a third monikette to the monikettes given "into/to".  
Add a fourth monikette to the monikettes given the other variable's type.  
Convert the monikettes to a moniker.  
Destroy the monikettes.  
Abort with "I don't know how to '" then the moniker then "'." and the locus.

198. We need this guy for help with expressions involving "then":

To convert a term for concatenation given a rider: \ used for right-side THEN operand  
If the term's variable is nil, abort with "I don't understand the variable '" then the term's phrase then "'." and the rider; exit.  
If the term's type can be reduced to "string", exit.  
Add an intermediate given "string" and the rider's token's first.  
Add the put or convert fragments given the term's variable and the intermediate and the rider's token's first.  
Put the intermediate into the term's variable.  
Clear the term's phrase.

199. And this is just another general-purpose helper:

To add a monikette to some monikettes given an expression:  
Create the monikette.  
Append the monikette to the monikettes.  
Put the expression's phrase into the monikette's string.  
Put the expression's variable into the monikette's variable.  
If the expression's variable is not nil, put the expression's type into the monikette's type.

200. This is the guy who adds a prolog fragment to every routine, together with his little helper:

To compile the body of a routine (prolog):  
If the routine is nil, exit.  
If the compiler's abort flag is set, exit.  
Add a fragment given the prolog tag.  
Add another fragment given the loop tag.

To add a fragment given a tag:  
If the current routine is nil, void the fragment; exit.  
Create the fragment given the tag.  
Append the fragment to the current routine's fragments.

201. Some sentences start with a Plain English keyword (like "if" or "loop"); others don't:

To compile the next statement given a rider:  
If the compiler's abort flag is set, exit.  
If the rider's token is "if", compile the next statement given the rider (if); exit.  
If the rider's token is "loop", compile the next statement given the rider (loop); exit.  
If the rider's token is "privatize", compile the next statement given the rider (privatize); exit.  
If the rider's token is not the start of any statement, abort with "I was expecting a statement here, but I found '" then the rider's token then "'." and the rider; exit.  
Compile the next statement given the rider (other).  
If the rider's token is not the period byte, abort with "You seem to have forgotten a period around here." and the rider; exit.  
Move the rider (compiler rules).

202. This is the guy who compiles IF statements, together with his two helpers:

To compile the next statement given a rider (if):  
If the compiler's abort flag is set, exit.  
Move the rider (compiler rules).  
Compile some monikettes given the rider.  
Remove any negatives from the monikettes returning a flag.  
Find a routine given the monikettes.  
If the routine is not nil, add push fragments given the monikettes.  
If the routine is nil, convert the monikettes to a moniker.  
Destroy the monikettes.  
If the routine is nil, abort with "I need a decider with the words '" then the moniker then "'." and the rider; exit.  
If the routine's decider flag is not set, abort with "Yeah, I'm going to need a decider here, ok?" and the rider; exit.  
If the rider's token is not the comma byte, abort with "I expected a comma around here, but I found '" then the rider's token then "'." and the rider; exit.  
Add a fragment given the call internal tag and the routine.  
If the flag is set, add another fragment given the not tag.  
Add a third fragment given the jump false tag.  
Move the rider (compiler rules).  
Loop.  
Compile the next statement given the rider (other).  
If the compiler's abort flag is set, exit.  
If the rider's token is the semi-colon byte, move the rider (compiler rules); repeat.  
If the rider's token is not the period byte, abort with "I'm thinking period, but you've got '" then the rider's token then "'." and the rider; exit.  
Move the rider (compiler rules).  
Add a fourth fragment given the end if tag.

To remove any negatives from some monikettes returning a flag:  
Clear the flag.  
Swap the monikettes with some other monikettes.  
Loop.  
Put the other monikettes' first into a monikette.  
If the monikette is nil, exit.  
Remove the monikette from the other monikettes.  
If the monikette's string is "not", reverse the flag; destroy the monikette; repeat.  
Append the monikette to the monikettes.  
If the monikette's string is "can't", reverse the flag; put "can" into the monikette's string; repeat.  
If the monikette's string is "cannot", reverse the flag; put "can" into the monikette's string; repeat.  
If the monikette's string is "nothing", reverse the flag; put "something" into the monikette's string; repeat.  
If the monikette's string is "won't", reverse the flag; put "will" into the monikette's string; repeat.  
If the monikette's string ends with "n't", reverse the flag; remove trailing bytes from the monikette's string given 3; repeat.  
Repeat.

To add push fragments given some monikettes:  
If the current routine is nil, exit.  
Loop.  
Get a monikette from the monikettes (backwards).  
If the monikette is nil, exit.  
If the monikette's variable is nil, repeat.  
If the monikette's current substring is not blank, repeat.  
If the monikette's current type is nil, repeat.  
Add a fragment given the push address tag and the monikette's variable.  
Repeat.

203. This is the guy who compiles other (ie, normal, programmer-created sentences):

To compile the next statement given a rider (other):  
If the compiler's abort flag is set, exit.  
If the rider's token is blank, exit.  
If the rider's token is "exit", compile the next statement given the rider (exit); exit.  
If the rider's token is "repeat", compile the next statement given the rider (repeat); exit.  
If the rider's token is "break", compile the next statement given the rider (break); exit.  
If the rider's token is "say", compile the next statement given the rider (say); exit.  
If the rider's token is "intel", compile the next statement given the rider (intel); exit.  
If the rider's token is "push", compile the next statement given the rider (push); exit.  
If the rider's token is "call", compile the next statement given the rider (call); exit.  
If the rider's token is "point", compile the next statement given the rider (point); exit.  
If the rider's token is "if", abort with "Nested 'if' statements are a sure sign of unclear thinking. Think again." and the rider; exit.  
If the rider's token is "privatize", abort with "The 'privatize' statement cannot be used here." and the rider; exit.  
If the rider's token is "loop", abort with "Looping inside an 'if' statement is a good way to lose your lunch." and the rider; exit.  
If the rider's token is "employ", abort with "'Employ' must be the first and only imperative in a routine." and the rider; exit.  
Compile the next statement given the rider (call - internal).

204. These are the four guys who compile EXIT and BREAK and SAY statements:

To compile the next statement given a rider (exit):  
If the current routine's decider flag is set, abort with "What does 'exit' mean here? Yes? No? Maybe?" and the rider; exit.  
Add a fragment given the exit tag.  
Move the rider (compiler rules).

To compile the next statement given a rider (repeat):  
Add a fragment given the repeat tag.  
Move the rider (compiler rules).

To compile the next statement given a rider (break):  
Add a fragment given the break tag.  
Move the rider (compiler rules).

To compile the next statement given a rider (say):  
If the current routine's decider flag is not set, compile the next statement given the rider (call - internal); exit.  
Move the rider (compiler rules).  
Compile an expression given the rider.  
If the compiler's abort flag is set, exit.  
If the expression's variable is nil, abort with "Invalid 'say', I don't understand the variable '" then the expression's phrase then "'." and the rider; exit.  
If the expression's type cannot be reduced to "flag", abort with "I need a definite 'yes' or 'no' here." and the rider; exit.  
Add a fragment given the load eax tag and the expression's variable.  
Add another fragment given the exit tag.

205. This is the guy who handles normal calls (that is, calls from one Plain English routine to another):

To compile the next statement given a rider (call - internal):  
Compile some monikettes given the rider.  
Find a routine given the monikettes.  
If the routine is not nil, add push fragments given the monikettes.  
If the routine is not nil, destroy the monikettes; add a fragment given the call internal tag and the routine; exit.  
Convert the monikettes to a moniker.  
Destroy the monikettes.  
Abort with "I don't know how to '" then the moniker then "'." and the rider.

206. This is the guy who handles inline machine code:

To compile the next statement given a rider (intel):  
Move the rider (compiler rules).  
If the rider's token is not any hex literal, abort with "I need some machine code here, I found '" then the rider's token then "'." and the rider; exit.  
Add a fragment given the intel tag.  
Put the rider's token into a nibble substring.  
Add 1 to the nibble substring's first.  
Convert the nibble substring to the fragment's code.  
Move the rider (compiler rules).

207. These two guys handle the PUSH statement, which is rarely used (except when you're writing a Plain English compiler from scratch):

To compile the next statement given a rider (push):  
If the compiler's abort flag is set, exit.  
Move the rider (compiler rules).  
Compile an expression given the rider.  
If the compiler's abort flag is set, exit.  
If the expression's variable is nil, abort with "Invalid 'push', I don't understand the variable '" then the expression's phrase then "'." and the rider; exit.  
If the expression's type is not value pushable, abort with "Values of type '" then the expression's type's name then "' can't be pushed." and the rider; exit.  
Add a fragment given the push value tag and the expression's variable.

To decide if a type is value pushable:  
If the type is nil, say no.  
If the type is nil, say no.  
If the type's length is 4, say yes.  
If the type's length is 2, say yes.  
If the type's length is 1, say yes.  
Say no.

208. These two guys handle abnormal calls (that is, calls to Windows functions):

To compile the next statement given a rider (call):  
If the compiler's abort flag is set, exit.  
Move the rider (compiler rules).  
If the rider's token is any string literal, compile the next statement given the rider (call - external); exit.  
Compile the next statement given the rider (call - indirect).

To compile the next statement given a rider (call - external):  
Put the rider's token into a string.  
Unquote the string.  
Move the rider (compiler rules).  
If the rider's token is not any string literal, abort with "Gee, I was expecting a quoted entry name, but I found '" then the rider's token then "'." and the rider; exit.  
Put the rider's token into another string.  
Unquote the other string.  
Move the rider (compiler rules).  
Add an entry to the imports given the string and the other string.  
If the rider's token is "with", compile the next statement given the rider (call - with clause).  
If the rider's token is "returning", compile the next statement given the rider returning a term (call - returning part).  
Add a fragment given the call external tag and the entry.  
If the term is empty, exit.  
If the term's variable is nil, abort with "I don't understand the variable '" then the term's phrase then "'." and the rider; exit.  
Add another fragment given the save eax tag and the term's variable.

209. A Dynamic Link Library (DLL) is a chunk of executable code that can be linked up with a program at run-time. All the Windows functions we use live in DLLs. Each DLL has to be "imported" at run-time. We have to keep track of all the DLLs (imports), and all the functions in those DLLs (entries), that a program calls. For example, this sentence...

Call "kernel32.dll" "Beep" with 220 and 200.

...refers to the "Beep" entry point in the "kernel32.dll". These five routines help us keep track of things like that:

To add an entry to some imports given an import name and an entry name:  
Find an import given the import name.  
If the import is nil, add the import to the imports given the import name.  
Find the entry given the entry name and the import's entries.  
If the entry is not nil, exit.  
Create the entry.  
Append the entry to the import's entries.  
Put the entry name into the entry's name.

To find an import given a string:  
Void the import.  
Loop.  
Get the import from the imports.  
If the import is nil, exit.  
If the import's name is the string, exit.  
Repeat.

To add an import to some imports given an import name:  
Create the import.  
Append the import to the imports.  
Put the import name into the import's name.

To create an import:  
Allocate memory for the import.

To find an entry given a string and some entries:  
Void the entry.  
Loop.  
Get the entry from the entries.  
If the entry is nil, exit.  
If the entry's name is the string, exit.  
Repeat.

To create an entry:  
Allocate memory for the entry.

210. For compatibility with Windows, we had to allow for certain ugly clauses in CALL statements. The WITH clause, for example, indicates parameters that are to be passed as values instead of addresses; and the RETURNING clause indicates where Windows should but the value he wants to return to us. These two routines deal with those clauses:

To compile the next statement given a rider (call - with clause): \ this guy is recursive so parameters get passed right to left  
If the compiler's abort flag is set, exit.  
Move the rider (compiler rules).  
Compile an expression given the rider.  
If the compiler's abort flag is set, exit.  
If the expression's variable is nil, abort with "I don't understand the variable '" then the expression's phrase then "'." and the rider; exit.  
If the expression's type is not value pushable, abort with "'" then the expression's type's name then "' can't be passed properly." and the rider; exit.  
If the rider's token is "and", compile the next statement given the rider (call - with clause).  
Add a fragment given the push value tag and the expression's variable.

To compile the next statement given a rider returning a term (call - returning part):  
Move the rider (compiler rules).  
Compile the term given the rider.  
If the compiler's abort flag is set, exit.  
If the term's variable is nil, abort with "I don't understand the variable '" then the term's phrase then "'." and the rider; exit.  
If the term's type is not value pushable, abort with "A simple 1, 2, or 4-byte variable is required in the 'returning' clause." and the rider.

211. And these are just two more helpers:

To add a fragment given a tag and an entry:  
If the current routine is nil, void the fragment; exit.  
Create the fragment given the tag.  
Append the fragment to the current routine's fragments.  
Put the entry into the fragment's entry.

To decide if a term is empty:  
If the term's variable is not nil, say no.  
If the term's phrase is not blank, say no.  
Say yes.

212. Sometimes Windows forces us to call a routine that is dynamically linked in a non-standard way, the address of the address of the routine being available to us only at run-time. This is the guy who handles that situation:

To compile the next statement given a rider (call - indirect):  
Compile an expression given the rider.  
If the compiler's abort flag is set, exit.  
If the expression's variable is nil, abort with "I don't understand the variable '" then the expression's phrase then "'." and the rider; exit.  
If the expression's type cannot be reduced to "pointer", abort with "I really need a pointer type in this CALL statement." and the rider; exit.  
If the rider's token is "with", compile the next statement given the rider (call - with clause).  
If the rider's token is "returning", compile the next statement given the rider returning a term (call - returning part).  
Add a fragment given the call indirect tag and the expression's variable.  
If the term is empty, exit.  
If the term's variable is nil, abort with "I don't understand the variable '" then the term's phrase then "'." and the rider; exit.  
Add another fragment given the save eax tag and the term's variable.

213. Sometimes Windows wants a pointer to one of our routines so he can call us in a non-standard and annoying way. For that, we need a special POINT command and this guy to compile it:

To compile the next statement given a rider (point):  
Move the rider (compiler rules).  
Compile a term given the rider.  
If the compiler's abort flag is set, exit.  
If the term's variable is nil, abort with "I don't understand the variable '" then the term's phrase then "'." and the rider; exit.  
If the term's kind is "literal", abort with "You've got the wrong kind of variable here." and the rider.  
If the term's type's length is not 4, abort with "You've got the wrong kind of variable here." and the rider.  
If the rider's token is not "to", abort with "I was expecting the word 'to', but found '" then the rider's token then "'." and the rider; exit.  
Move the rider (compiler rules).  
If the rider's token is not "routine", abort with "I was expecting the word 'routine' but found '" then the rider's token then "'." and the rider; exit.  
Move the rider (compiler rules).  
Compile a routine reference given the rider.  
Convert the routine reference to a moniker.  
Destroy the routine reference.  
Find a routine given the moniker and the routine index.  
If the routine is nil, abort with "I can't find the '" then the moniker then "' routine to which you are trying to point." and the rider; exit.  
Add a fragment given the routine address tag and the routine.  
Put the term's variable into the fragment's variable.

214. This is the guy who compiles LOOP statements:

To compile the next statement given a rider (loop):  
If the compiler's abort flag is set, exit.  
Add a fragment given the loop tag.  
Move the rider (compiler rules).  
If the rider's token is not the period byte, abort with "You seem to have forgotten a period around here." and the rider; exit.  
Move the rider (compiler rules).

215. And this is the guy who compiles PRIVATIZE statements:

To compile the next statement given a rider (privatize):  
If the compiler's abort flag is set, exit.  
Move the rider (compiler rules).  
Compile a term given the rider.  
If the compiler's abort flag is set, exit.  
If the term's variable is nil, abort with "Invalid 'privatize', I don't understand the variable '" then the term's phrase then "'." and the rider; exit.  
If the term's kind is not "parameter", abort with "You can only privatize parameters." and the rider; exit.  
Add a local to the current routine's locals given "local" and the rider's token's first.  
Put the term's name into the local's name.  
Put the term's nickname into the local's nickname.  
Put the term's type name into the local's type name.  
Resolve the local.  
Prepend "original " to the term's name.  
If the term's nickname is not blank, prepend "original " to the term's nickname.  
Add several fragments given "put" and the term's variable and "into" and the local and the rider's token's first.  
If the rider's token is not the period byte, abort with "You seem to have forgotten a period around here." and the rider; exit.  
Move the rider (compiler rules).

216. This is just another general-purpose helper:

To decide if a string is the start of any statement:  
If the string is the start of any definition, say no.  
If the string is any literal, say no.  
If the string is any mark, say no.  
If the string is blank, say no.  
Say yes.

217. And these are the two guys we need to put an EPILOG into every routine (to clean up the stack, restore registers, and dump any Heap memory that is no longer needed):

To compile the body of a routine (epilog):  
If the routine is nil, exit.  
If the compiler's abort flag is set, exit.  
Add a fragment given the finalize tag.  
Loop.  
Get a variable from the routine's locals.  
If the variable is nil, break.  
If the variable's kind is "scratch", repeat.  
If the variable's type should not be finalized, repeat.  
Add another fragment given the push address tag and the variable.  
Find another routine given "~finalize" and the variable's type.  
If the other routine is nil, abort with "Internal error - compile the body of a routine (epilog)"; exit.  
Add a third fragment given the call internal tag and the other routine.  
Repeat.  
Add a fourth fragment given the epilog tag.

To find a routine given a string and a type:  
Add a monikette to some monikettes given the string.  
Add another monikette to the monikettes given the type.  
Find the routine given the monikettes.  
Destroy the monikettes.  
  
218. EMPLOY and POINT statements explicitly refer to other routines, and some terms implicitly refer to other routines (like "a box's center" which is an implicit reference to the function-routine "To put a box's center into a spot.") A *routine reference* is a list of monikettes that refers to such a routine, and these are the routines we use to compile them:

To compile a routine reference given a rider:  
If the rider's token is "to", move the rider (compiler rules).  
If the rider is the start of any function, compile the routine reference given the rider (function); exit.  
If the rider's token is "decide", compile the routine reference given the rider (decide).  
Loop.  
If the compiler's abort flag is set, exit.  
If the rider's token is blank, exit.  
If the rider's token is ".", exit.  
If the rider's token is "the", compile the routine reference given the rider (definite article); repeat.  
If the rider's token is any indefinite article, compile the routine reference given the rider (indefinite article); repeat.  
If the rider's token is any monikette string, compile the routine reference given the rider (monikette string); repeat.  
Abort with "'" then the rider's token then "' is not valid in an employ statement." and the rider.

To compile a routine reference given a rider (decide):  
Move the rider (compiler rules).  
If the rider's token is not "if", abort with "We always say DECIDE IF, not DECIDE and then '" then the rider's token then "'." and the rider; exit.  
Move the rider (compiler rules).

To compile a routine reference given a rider (definite article):  
If the compiler's abort flag is set, exit.  
Move the rider (compiler rules).  
Scan a name given the rider.  
Prepend "the " to the name.  
Add a monikette to the routine reference given the name.

To compile a routine reference given a rider (function):  
If the compiler's abort flag is set, exit.  
Add a monikette to the routine reference given "put".  
Move the rider (compiler rules).  
If the rider's token is any indefinite article, set a flag.  
If the flag is set, compile the routine reference given the rider (indefinite article).  
If the flag is not set, compile the routine reference given the rider (definite article).  
If the rider's token is not any possessive, abort with "I was expecting an 's, but found '" then the rider's token then "'." and the rider; exit.  
Move the rider (compiler rules).  
Scan a name given the rider.  
Add another monikette to the routine reference given "'s " then the name.  
If the rider's token is not "into", abort with "I was expecting the word 'into', but found '" then the rider's token then "'." and the rider; exit.  
Add a third monikette to the routine reference given "into".  
Move the rider (compiler rules).  
If the rider's token is not any indefinite article, abort with "I was expecting an indefinite article, but found '" then the rider's token then "'." and the rider; exit.  
Compile the routine reference given the rider (indefinite article).

To compile a routine reference given a rider (indefinite article):  
If the compiler's abort flag is set, exit.  
Move the rider (compiler rules).  
Scan a name given the rider.  
Find a type and a nickname given the name.  
If the type is nil, abort with "Invalid employ statement, I don't understand '" then the name then "'." and the rider; exit.  
Add a monikette to the routine reference given the type.

To compile a routine reference given a rider (monikette string):  
If the compiler's abort flag is set, exit.  
Add a monikette to the routine reference given the rider's token.  
Move the rider (compiler rules).  
  
219. Whew! Made it. Run it, open the daughter in the blue, and List her. Then open the listing, find "ROUTINES:" (no quotes) and you'll see a lot of scratches, intermediates, and fragments hanging on our routines. I've inserted some source lines in the listing below, in blue, to make the fragments easier to understand (the fragments generated by each statement appear below the source):

ROUTINES:

To abort with a string:  
/routine/abort from/given/with/using [string]/yes/no/no/no//0/0/00000000/  
/variable/parameter/yes/string/string/string/string/00000000/no/1/no////  
/variable/scratch/yes/~S7063//flag/flag/00000000/no/1/no////  
/variable/scratch/yes/~S7064//flag/flag/00000000/no/1/no////  
/variable/scratch/yes/~S7065//abort message/abort message/00000000/no/1/no////  
/fragment/prolog/////00000000/00000000//  
/fragment/loop/////00000000/00000000//

If the compiler's abort flag is set, exit.  
/fragment/load address/compiler/~S7063///00000000/00000000//  
/fragment/increment/~S7063////00000014/00000000//  
/fragment/push address/~S7063////00000000/00000000//  
/fragment/call internal///[flag] is/are set//00000000/00000000//  
/fragment/jump false/////00000000/00000000//  
/fragment/exit/////00000000/00000000//  
/fragment/end if/////00000000/00000000//

Set the compiler's abort flag.  
/fragment/load address/compiler/~S7064///00000000/00000000//  
/fragment/increment/~S7064////00000014/00000000//  
/fragment/push address/~S7064////00000000/00000000//  
/fragment/call internal///set [flag]//00000000/00000000//

Put the string into the compiler's abort message.  
/fragment/load address/compiler/~S7065///00000000/00000000//  
/fragment/increment/~S7065////00000018/00000000//  
/fragment/push address/~S7065////00000000/00000000//  
/fragment/push address/string////00000000/00000000//  
/fragment/call internal///put [string] in/into/to [string]//00000000/00000000//  
/fragment/finalize/////00000000/00000000//  
/fragment/epilog/////00000000/00000000//

At the bottom of the listing you'll see that this is the part of compiling that really eats up the time:

TIMERS:

/timer/loading/0/  
/timer/scanning/515/  
/timer/resolving types/32/  
/timer/resolving globals/109/  
/timer/compiling routine headers/281/  
/timer/calculating/16/  
/timer/adding built-in memory routines/47/  
/timer/indexing utilities/94/  
/timer/compiling routine bodies/1468/  
/timer/adding built-in startup routine/0/  
/timer/offsetting/0/  
/timer/addressing/0/  
/timer/transmogrifying/0/  
/timer/linking/0/  
/timer/writing/0/  
/timer/total/5187/

Almost home now. The rest may be tedious, but it's easy stuff.

---

Now that we know everything in the source checks out, it's time to add one more routine of our own.

220. Move our abort line down a notch and activate the next step:

To compile a directory:  
Compile the directory (start).  
Compile the directory (load the source files).  
Compile the directory (scan the source files).  
Compile the directory (resolve the types).  
Compile the directory (resolve the globals).  
Compile the directory (compile the headers of the routines).  
Compile the directory (calculate lengths and offsets of types).  
Compile the directory (add the built-in memory routines).  
Compile the directory (index the routines for utility use).  
Compile the directory (compile the bodies of the routines).  
Compile the directory (add and compile the built-in startup routine).  
Abort with "I'm not old enough to make babies.".  
\Compile the directory (offset parameters and variables).  
\Compile the directory (address).  
\Compile the directory (transmogrify).  
\Compile the directory (link).  
\Compile the directory (write the exe).  
\Compile the directory (stop).

221. Then add this guy back in:

To compile a directory (add and compile the built-in startup routine):  
If the compiler's abort flag is set, exit.  
Show status "Adding...".  
Start the adding built-in startup routine timer.  
Add and compile the initialize before run and run and finalize after run routine.  
Stop the adding built-in startup routine timer.

222. And this guy, who creates the routine that will be the first routine to get control when the programmer's program is run:

To add and compile the initialize before run and run and finalize after run routine:  
If the compiler's abort flag is set, exit.  
Add a routine to the routines given nil.  
Append "~initialize before run and run and finalize after run" to the routine's header string.  
Append "initialize before run. " to the routine's body string.  
Append the global initializers to the routine's body string.  
Append "run. " to the routine's body string.  
Append the global finalizers to the routine's body string.  
Append "finalize after run. " to the routine's body string.  
Append "call ""kernel32.dll"" ""ExitProcess"" with 0. " to the routine's body string.  
Slap the routine's routine header on the routine's header string.  
Slap the routine's routine body on the routine's body string.  
Compile the header of the routine.  
Compile the body of the routine.

223. Then uncomment these two guys, who add initializer and finalizer code for global variables to our start-up routine:

To append the global initializers to a string:  
Get a global from the globals.  
If the global is nil, exit.  
If the global's literal is nil, repeat.  
If the global's compiled flag is not set, repeat.  
Clear a flag.  
Append the global's initializer string to the string.  
Repeat.

To append the global finalizers to a string:  
Get a global from the globals.  
If the global is nil, exit.  
If the global's compiled flag is not set, repeat.  
If the global's type should not be finalized, repeat.  
Append "~finalize the " then the global's name then ". " to the string.  
Repeat.

224. And that's it for this step. Run it. In the blue, open the daughter and list her. Then open the listing and look for "run and run" (without the quotes); that will take you to the routine we just added, which should look like this (it's a biggie, so I've added some comments in blue and cut out a lot of repetitive lines):

/routine/~initialize before run and run and finalize after run/yes/no/no/no//0/0/00000000/  
/fragment/prolog/////00000000/00000000//  
/fragment/loop/////00000000/00000000//

/fragment/call internal///initialize before run//00000000/00000000//  
\ global initializers start here  
/fragment/push address/break tag////00000000/00000000//  
/fragment/push address/~L1////00000000/00000000//  
/fragment/call internal///put [number] in/into/to [number]//00000000/00000000//  
/fragment/push address/call external tag////00000000/00000000//  
/fragment/push address/~L2////00000000/00000000//  
/fragment/call internal///put [number] in/into/to [number]//00000000/00000000//  
\ blah, blah, blah

/fragment/push address/document version////00000000/00000000//  
/fragment/push address/~L397////00000000/00000000//  
/fragment/call internal///put [string] in/into/to [string]//00000000/00000000//  
/fragment/push address/edge height////00000000/00000000//

\ global initializers end here  
/fragment/call internal///run//00000000/00000000//

\ global finalizers start here  
/fragment/push address/compiler////00000000/00000000//  
/fragment/call internal///~finalize [~inline compiler type]//00000000/00000000//  
/fragment/push address/answer////00000000/00000000//  
/fragment/call internal///~finalize [string]//00000000/00000000//

\ blah, blah, blah  
/fragment/push address/temp path////00000000/00000000//  
/fragment/call internal///~finalize [string]//00000000/00000000//  
/fragment/push address/document version////00000000/00000000//  
/fragment/call internal///~finalize [string]//00000000/00000000//

/fragment/push address/remembered picture path////00000000/00000000//  
/fragment/call internal///~finalize [string]//00000000/00000000//  
\ global finalizers end here  
/fragment/call internal///finalize after run//00000000/00000000//  
/fragment/push value/~L12694////00000000/00000000//  
/fragment/call external////ExitProcess/00000000/00000000//  
/fragment/finalize/////00000000/00000000//  
/fragment/epilog/////00000000/00000000//

Next time, we'll calculate the offsets for our parameters and local variables.

---

225. Time to calculate some offsets. Let's get right to it. Move our abort line down and reinstate the next step:

To compile a directory:  
Compile the directory (start).  
Compile the directory (load the source files).  
Compile the directory (scan the source files).  
Compile the directory (resolve the types).  
Compile the directory (resolve the globals).  
Compile the directory (compile the headers of the routines).  
Compile the directory (calculate lengths and offsets of types).  
Compile the directory (add the built-in memory routines).  
Compile the directory (index the routines for utility use).  
Compile the directory (compile the bodies of the routines).  
Compile the directory (add and compile the built-in startup routine).  
Compile the directory (offset parameters and variables).  
Abort with "I'm not old enough to make babies.".  
\Compile the directory (address).  
\Compile the directory (transmogrify).  
\Compile the directory (link).  
\Compile the directory (write the exe).  
\Compile the directory (stop).

226. Then put this guy back:

To compile a directory (offset parameters and variables):  
If the compiler's abort flag is set, exit.  
Show status "Offsetting...".  
Start the offsetting timer.  
Offset the parameters in the routines.  
Offset the locals in the routines.  
Stop the offsetting timer.

227. Now these two guys go back in:

To offset the parameters in some routines:  
If the compiler's abort flag is set, exit.  
Get a routine from the routines.  
If the routine is nil, exit.  
Offset the parameters in the routine.  
Repeat.

To offset the parameters in a routine:  
If the routine is nil, exit.  
If the routine's compiled flag is not set, exit.  
Clear the routine's parameter size.  
Loop.  
If the compiler's abort flag is set, exit.  
Get a parameter from the routine's parameters.  
If the parameter is nil, exit.  
If the routine's callback flag is set, set the parameter's by-value flag.  
If the parameter's previous is nil, put 8 into the parameter's offset. \ skip the return address and saved ebp  
If the parameter's previous is not nil, put the parameter's previous' offset plus 4 into the parameter's offset. \ all parameters are 4 bytes long

Add 4 to the routine's parameter size.  
Repeat.

228. And finally these two guys:

To offset the locals in some routines:  
If the compiler's abort flag is set, exit.  
Get a routine from the routines.  
If the routine is nil, exit.  
Offset the locals in the routine.  
Repeat.

To offset the locals in a routine:  
If the routine is nil, exit.  
If the routine's compiled flag is not set, exit.  
Clear the routine's local size.  
Loop.  
If the compiler's abort flag is set, exit.  
Get a local from the routine's locals.  
If the local is nil, exit.  
If the local's previous is not nil, put the local's previous' offset into the local's offset.  
Put the local's type's length into a length.  
If the local's kind is "scratch", put a pointer's magnitude into the length.  
Round the length up to the nearest multiple of 4.  
Subtract the length from the local's offset.  
Add the length to the routine's local size.  
Repeat.

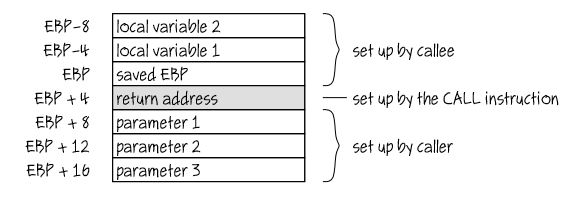
229. Now run it. In the blue, open up the daughter and list her. Then get into the listing and look for "routines:" (with the colon, no quotes). You should see something that starts like this:

ROUTINES:

/routine/abort from/given/with/using [string]/yes/no/no/no//4/12/00000000/  
/variable/parameter/yes/string/string/string/string/00000008/no/1/no////  
/variable/scratch/yes/~S7063//flag/flag/FFFFFFFC/no/1/no////  
/variable/scratch/yes/~S7064//flag/flag/FFFFFFF8/no/1/no////  
/variable/scratch/yes/~S7065//abort message/abort message/FFFFFFF4/no/1/no////

/fragment/prolog/////00000000/00000000//

Now recall that the stack, in general, is arranged like this:



Note the *offset* on our parameter: 00000008. That means 8 bytes past wherever the EBP register is pointing.

Note also the offsets on the local *scratch* variables: FFFFFFFC, FFFFFFF8, FFFFFFF4. That's -4, -8, and -12 in regular notation, which means 4, 8, and 12 bytes to the other side of where the EBP register is pointing.

And that's it for this step. Next time, we'll give everyone (who needs one) a permanent address.

---

230. Time to hand out addresses. Move the abort line down and activate the next step:

To compile a directory:  
Compile the directory (start).  
Compile the directory (load the source files).  
Compile the directory (scan the source files).  
Compile the directory (resolve the types).  
Compile the directory (resolve the globals).  
Compile the directory (compile the headers of the routines).  
Compile the directory (calculate lengths and offsets of types).  
Compile the directory (add the built-in memory routines).  
Compile the directory (index the routines for utility use).  
Compile the directory (compile the bodies of the routines).  
Compile the directory (add and compile the built-in startup routine).  
Compile the directory (offset parameters and variables).  
Compile the directory (address).  
Abort with "I'm not old enough to make babies.".  
\Compile the directory (transmogrify).  
\Compile the directory (link).  
\Compile the directory (write the exe).  
\Compile the directory (stop).

231. Then put this guy back in:

To compile a directory (address):  
If the compiler's abort flag is set, exit.  
Show status "Addressing...".  
Start the addressing timer.  
Put 4096 into an address.  
Put the address into the import borough's base.  
Address the imports given the address.  
Put the address minus the import borough's base into the import borough's length.  
Round the address up to the nearest multiple of 4096.  
Put the address into the data borough's base.  
Address the globals given the address.  
Address the literals given the address.  
Put the address minus the data borough's base into the data borough's length.  
Round the address up to the nearest multiple of 4096.  
Put the address into the code borough's base.  
Address the routines given the address.  
Put the address minus the code borough's base into the code borough's length.  
Stop the addressing timer.

232. You hopefully recall that we only need five of the many possible sections that a Windows Portable Executable might have: The DOS Header, the PE Header, the Import Section, the Data Section, and the Code Section. We'll deal with the two Headers later. In the meantime, we didn't want to waste a good word like "section" on Windows' crap, so we decided to call those latter three sections "boroughs." (A *borough* is an administrative division of a large city.) Here are the types we'll be needing in this step; put them back in:

A borough has a base address, a length and a size.

The import borough is a borough.

The data borough is a borough.

The code borough is a borough.

The *size* of a borough is its actual *length* rounded up to the nearest multiple of 4096 (the virtual page size used by x86 architecture CPUs).

233. We'll also be needing a few globals to keep track of where we are; put these back in too:

The current header address is an address.

The current name address is an address.

The current thunk address is an address.

The image base is an address equal to 4194304 [$00400000].

234. The Import Borough comes just after the headers in our executable, so we address our imports first:

To address some imports given an address:  
Put the address into the current header address.  
Put the imports' count into a count.  
Add 1 to the count.  
Multiply the count by an import header's magnitude.  
Put the address plus the count into the current thunk address.  
Get another count given the imports (all entries plus markers).  
Multiply the other count by 4.  
Put the current thunk address plus the other count into the current name address.  
Loop.  
Get an import from the imports.  
If the import is nil, break.  
Address the import.  
Repeat.  
Put the current name address minus the address into a number.  
Add the number to the address.

235. The routine above needs a little help from this guy:

To get a count given some imports (all entries plus markers):  
Put 0 into the count.  
Loop.  
Get an import from the imports.  
If the import is nil, exit.  
Add the import's entries' count to the count.  
Add 1 to the count.  
Repeat.

236. And some more help from this routine:

To address an import:  
Put the current name address into the import's name address.  
Put the current header address into the import's header address.  
Put the current name address into the import's import header's name memory address.  
Put the import's name's length plus 1 into a number.  
Round the number up to the nearest multiple of 2.  
Add the number to the current name address.  
Add an import header's magnitude to the current header address.  
Put the current thunk address into the import's import header's first thunk memory address.  
Address the import's entries.  
Add 4 to the current thunk address.

237. it's not enough, of course, to just address our imports (the DLLs we'll be calling); we need to address each of the functions (entries) within those DLLs that we want to call. So we need these two guys:

To address some entries:  
Get an entry from the entries.  
If the entry is nil, exit.  
Address the entry.  
Repeat.

To address an entry:  
Put the current name address into the entry's name address.  
Put the entry's name's length plus 3 into a number.  
Round the number up to the nearest multiple of 2.  
Add the number to the current name address.  
Put the current thunk address into the entry's thunk address.  
Put the image base plus the current thunk address into the entry's address.  
Add 4 to the current thunk address.

238. And that's it for the DLL stuff. Now we assign addresses to our global variables and our literals, both using these two guys:

To address some variables given an address:  
If the compiler's abort flag is set, exit.  
Get a variable from the variables.  
If the variable is nil, exit.  
Address the variable given the address.  
Repeat.

To address a variable given an address:  
If the compiler's abort flag is set, exit.  
If the variable is nil, exit.  
If the variable's compiled flag is not set, exit.  
Put the image base plus the address into the variable's address.  
Add the variable's type's length to the address.  
If the variable's type can be reduced to "string", add the variable's data's length plus 1 to the address.  
Round the address up to the nearest multiple of 4.

239. Lastly, we assign addresses to each of our routines:

To address some routines given an address:  
If the compiler's abort flag is set, exit.  
Get a routine from the routines.  
If the routine is nil, break.  
Address the routine given the address.  
Repeat.

To address a routine given an address:  
If the compiler's abort flag is set, exit.  
If the routine is nil, exit.  
If the routine's employs moniker is not blank, exit. \ employs are addressed later  
If the routine's compiled flag is not set, exit.  
Put the routine into the current routine.  
Put the image base plus the address into the routine's address.  
Address the routine's fragments given the address.  
Round the address up to the nearest multiple of 4.

240. it's not enough, of course, to just give an address to a routine's entry point; we need to know where to put each fragment of code, as well. So we need these two guys:

To address some fragments given an address:  
If the compiler's abort flag is set, exit.  
Get a fragment from the fragments.  
If the fragment is nil, exit.  
Address the fragment given the address.  
Repeat.

To address a fragment given an address:  
If the compiler's abort flag is set, exit.  
If the fragment is nil, exit.  
Put the image base plus the address into the fragment's address.  
If the fragment's tag is the push address tag, add 7 to the address; exit.  
If the fragment's tag is the call internal tag, add 5 to the address; exit.  
If the fragment's tag is the load address tag, add 12 to the address; exit.  
If the fragment's tag is the increment tag, add 10 to the address; exit.  
If the fragment's tag is the dereference tag, add 14 to the address; exit.  
If the fragment's tag is the jump false tag, add 9 to the address; exit.  
If the fragment's tag is the not tag, add 3 to the address; exit.  
If the fragment's tag is the exit tag, add 5 to the address; exit.  
If the fragment's tag is the repeat tag, add 5 to the address; exit.  
If the fragment's tag is the break tag, add 5 to the address; exit.  
If the fragment's tag is the prolog tag, address the fragment given the address (prolog); exit.  
If the fragment's tag is the epilog tag, address the fragment given the address (epilog); exit.  
If the fragment's tag is the intel tag, add the fragment's code's length to the address; exit.  
If the fragment's tag is the push value tag, address the fragment given the address (push value); exit.  
If the fragment's tag is the call external tag, add 6 to the address; exit.  
If the fragment's tag is the load eax tag, add 8 to the address; exit.  
If the fragment's tag is the save eax tag, address the fragment given the address (save eax); exit.  
If the fragment's tag is the call indirect tag, add 8 to the address; exit.  
If the fragment's tag is the routine address tag, add 12 to the address; exit.

241. Some fragments have fixed machine-code lengths; some don't. The fixed-length ones are handled above; the others are handled by the four routines below:

To address a fragment given an address (epilog):  
If the current routine's callback flag is set, add 3 to the address.  
Add 6 to the address.

To address a fragment given an address (prolog):  
Add 3 to the address.  
If the current routine's local size is not 0, add 10 to the address.  
If the current routine's callback flag is set, add 3 to the address.

To address a fragment given an address (push value):  
Add 6 to the address.  
If the fragment's variable is nil, abort with "Internal error - address a fragment given an address (push value)"; exit.  
If the fragment's variable's type is nil, abort with "Internal error 2 - address a fragment given an address (push value)"; exit.  
Put the fragment's variable's type's length into a length.  
If the length is 4, add 2 to the address; exit.  
If the length is 2, add 3 to the address; exit.  
If the length is 1, add 4 to the address; exit.  
Abort with "Internal error 3 - address a fragment given an address (push value)".

To address a fragment given an address (save eax):  
Add 6 to the address.  
If the fragment's variable is nil, abort with "Internal error - address a fragment given an address (push value)"; exit.  
If the fragment's variable's type is nil, abort with "Internal error 2 - address a fragment given an address (push value)"; exit.  
Put the fragment's variable's type's length into a length.  
If the length is 4, add 2 to the address; exit.  
If the length is 2, add 3 to the address; exit.  
If the length is 1, add 2 to the address; exit.  
Abort with "Internal error 3 - address a fragment given an address (push value)".

242. And that's it for this step. Run it. In the blue, open up the daughter and list her. Then open up the listing and check out the addresses we've given our globals:

GLOBALS:

/variable/global/yes/adding built-in memory routines timer//timer/timer/00402000/no/1/no////  
/variable/global/yes/adding built-in startup routine timer//timer/timer/0040200C/no/1/no////  
/variable/global/yes/addressing timer//timer/timer/00402018/no/1/no////

Then jump to the literals and check out their addresses:

LITERALS:

/variable/literal/yes/~L1//number/number/004030AC/no/1/no///01000000/  
/variable/literal/yes/~L2//number/number/004030B0/no/1/no///02000000/  
/variable/literal/yes/~L3//number/number/004030B4/no/1/no///03000000/

Then jump to the routines and check out both the entry addresses, and the fragment addresses:

ROUTINES:

/routine/abort from/given/with/using [string]/yes/no/no/no//4/12/00419000/  
/variable/parameter/yes/string/string/string/string/00000008/no/1/no////  
/variable/scratch/yes/~S7063//flag/flag/FFFFFFFC/no/1/no////  
/variable/scratch/yes/~S7064//flag/flag/FFFFFFF8/no/1/no////  
/variable/scratch/yes/~S7065//abort message/abort message/FFFFFFF4/no/1/no////  
/fragment/prolog/////00000000/00419000//  
/fragment/loop/////00000000/0041900D//  
/fragment/load address/compiler/~S7063///00000000/0041900D//  
/fragment/increment/~S7063////00000014/00419019//  
/fragment/push address/~S7063////00000000/00419023//  
/fragment/call internal///[flag] is/are set//00000000/0041902A//  
/fragment/jump false/////00000000/0041902F//  
/fragment/exit/////00000000/00419038//  
/fragment/end if/////00000000/0041903D//  
/fragment/load address/compiler/~S7064///00000000/0041903D//  
/fragment/increment/~S7064////00000014/00419049//  
/fragment/push address/~S7064////00000000/00419053//  
/fragment/call internal///set [flag]//00000000/0041905A//  
/fragment/load address/compiler/~S7065///00000000/0041905F//  
/fragment/increment/~S7065////00000018/0041906B//  
/fragment/push address/~S7065////00000000/00419075//  
/fragment/push address/string////00000000/0041907C//  
/fragment/call internal///put [string] in/into/to [string]//00000000/00419083//  
/fragment/finalize/////00000000/00419088//  
/fragment/epilog/////00000000/00419088//

Finally, jump to "IMPORTS:" (no quotes) and take a peek at both the import and entry addresses we assigned:

IMPORTS:

/import/user32.dll/  
/entry/DisableProcessWindowsGhosting/004010DC/  
/entry/GetSystemMetrics/004010E0/  
/entry/PostMessageA/004010E4/

Sweet. And all that addressing took only... let me see...

TIMERS:

/timer/loading/16/  
/timer/scanning/437/  
/timer/resolving types/15/  
/timer/resolving globals/110/  
/timer/compiling routine headers/281/  
/timer/calculating/0/  
/timer/adding built-in memory routines/62/  
/timer/indexing utilities/79/  
/timer/compiling routine bodies/1437/  
/timer/adding built-in startup routine/31/  
/timer/offsetting/0/  
/timer/addressing/32/  
/timer/transmogrifying/0/  
/timer/linking/0/  
/timer/writing/0/

/timer/total/5078/

...32 milliseconds. Zippy!

Next up: TRANSMOGRIFICATION! Mwa-hah-hah-hah (evil laugh).

---

243. The dictionary says *transmogrify* means "to change completely the nature or appearance of," and that's exactly what we're about to do to each and every one of our fragments. So let's do it. Move the abort line down, reactivate the next step:

To compile a directory:  
Compile the directory (start).  
Compile the directory (load the source files).  
Compile the directory (scan the source files).  
Compile the directory (resolve the types).  
Compile the directory (resolve the globals).  
Compile the directory (compile the headers of the routines).  
Compile the directory (calculate lengths and offsets of types).  
Compile the directory (add the built-in memory routines).  
Compile the directory (index the routines for utility use).  
Compile the directory (compile the bodies of the routines).  
Compile the directory (add and compile the built-in startup routine).  
Compile the directory (offset parameters and variables).  
Compile the directory (address).  
Compile the directory (transmogrify).  
Abort with "I'm not old enough to make babies.".  
\Compile the directory (link).  
\Compile the directory (write the exe).  
\Compile the directory (stop).

244. Then add in these three guys:

To compile a directory (transmogrify):  
If the compiler's abort flag is set, exit.  
Show status "Mogrifying...".  
Start the transmogrifying timer.  
Transmogrify the routines.  
Stop the transmogrifying timer.

To transmogrify some routines:  
If the compiler's abort flag is set, exit.  
Get a routine from the routines.  
If the routine is nil, break.  
Transmogrify the routine.  
Repeat.

To transmogrify a routine:  
If the compiler's abort flag is set, exit.  
If the routine is nil, exit.  
If the routine's employs moniker is not blank, exit.  
If the routine's compiled flag is not set, exit.  
Put the routine into the current routine.  
Transmogrify the routine's fragments.

245. These two routines are the top-level helpers we need:

To transmogrify some fragments:  
If the compiler's abort flag is set, exit.  
Get a fragment from the fragments.  
If the fragment is nil, exit.  
Transmogrify the fragment.  
Repeat.

To transmogrify a fragment:  
If the compiler's abort flag is set, exit.  
If the fragment is nil, exit.  
If the fragment's tag is the push address tag, transmogrify the fragment (push address); exit.  
If the fragment's tag is the call internal tag, transmogrify the fragment (call internal); exit.  
If the fragment's tag is the load address tag, transmogrify the fragment (load address); exit.  
If the fragment's tag is the increment tag, transmogrify the fragment (increment); exit.  
If the fragment's tag is the dereference tag, transmogrify the fragment (dereference); exit.  
If the fragment's tag is the jump false tag, transmogrify the fragment (jump false); exit.  
If the fragment's tag is the not tag, transmogrify the fragment (not); exit.  
If the fragment's tag is the exit tag, transmogrify the fragment (exit); exit.  
If the fragment's tag is the repeat tag, transmogrify the fragment (repeat); exit.  
If the fragment's tag is the break tag, transmogrify the fragment (break); exit.  
If the fragment's tag is the prolog tag, transmogrify the fragment (prolog); exit.  
If the fragment's tag is the epilog tag, transmogrify the fragment (epilog); exit.  
If the fragment's tag is the push value tag, transmogrify the fragment (push value); exit.  
If the fragment's tag is the call external tag, transmogrify the fragment (call external); exit.  
If the fragment's tag is the load eax tag, transmogrify the fragment (load eax); exit.  
If the fragment's tag is the save eax tag, transmogrify the fragment (save eax); exit.  
If the fragment's tag is the call indirect tag, transmogrify the fragment (call indirect); exit.  
If the fragment's tag is the routine address tag, transmogrify the fragment (routine address); exit.

246. And these 18 guys are the low-level helpers who actually do the transmogrification:

To transmogrify a fragment (break):  
Put the repeat tag into a tag.  
Find another fragment given the fragment and the tag.  
If the other fragment is nil, put the finalize tag into the tag. \ for break without a loop or after a loop  
Find a third fragment given the current routine's fragments' last and the tag (backwards).  
If the third fragment is nil, abort with "Internal error - transmogrify a fragment (break)"; exit.  
If the third fragment's next is nil, abort with "Internal error 2 - transmogrify a fragment (break)"; exit.  
Attach $E9 and the third fragment's next's address to the fragment. \ jmp the destination's address

To transmogrify a fragment (call external):  
Attach $FF15 and the fragment's entry's address to the fragment's code. \ call [the fragment's entry's address]

To transmogrify a fragment (call indirect):  
Attach address loading code to the fragment's code given the fragment's variable. \ put address into edx  
Attach $FF12 to the fragment's code. \ call [edx]

To transmogrify a fragment (call internal):  
Get an address given the fragment's routine.  
Attach $E8 and the address to the fragment. \ call the fragment's routine's address

To transmogrify a fragment (dereference):  
Attach $8B95 and the fragment's variable's offset to the fragment's code. \ mov edx,[ebp+the fragment's variable's offset]  
Attach $8B12 to the fragment's code. \ mov edx,[edx]  
Attach $8995 and the fragment's variable's offset to the fragment's code. \ mov [ebp+the fragment's variable's offset],edx

To transmogrify a fragment (epilog):  
If the current routine's callback flag is set, attach $5F5E5B to the fragment's code. \ pop edi, esi, ebx  
Attach $8BE5 to the fragment's code. \ mov esp,ebp  
Attach $5D to the fragment's code. \ pop ebp  
Attach $C2 and the current routine's parameter size to the fragment's code. \ ret the current routine's parameter size

To transmogrify a fragment (exit):  
Find another fragment given the fragment and the finalize tag.  
If the other fragment is nil, abort with "Internal error - transmogrify a fragment (exit)"; exit.  
Attach $E9 and the other fragment's address to the fragment. \ jmp the destination's address

To transmogrify a fragment (increment):  
If the fragment's variable is nil, abort with "Internal error - transmogrify a fragment (increment)"; exit.  
If the fragment's variable's kind is not "scratch", abort with "Internal error 2 - transmogrify a fragment (increment)"; exit.  
Attach $8185 and the fragment's variable's offset and the fragment's number to the fragment's code. \ add [ebp+the fragment's variable's offset],the fragment's number

To transmogrify a fragment (jump false):  
Find another fragment given the fragment and the end if tag.  
If the other fragment is nil, abort with "Internal error - transmogrify a fragment (jump false)"; exit.  
Attach $83F800 to the fragment's code. \ cmp eax,0  
Attach $0F84 and the other fragment's address to the fragment. \ je the destination's address

To transmogrify a fragment (load address):  
Attach address loading code to the fragment's code given the fragment's variable. \ put address into edx  
Attach $8995 and the fragment's other variable's offset to the fragment's code. \ mov [ebp+the fragment's variable's offset],edx

To transmogrify a fragment (load eax):  
Attach address loading code to the fragment's code given the fragment's variable. \ put address into edx  
If the fragment's variable is nil, abort with "Internal error - transmogrify a fragment (load eax)"; exit.  
If the fragment's variable's type is nil, abort with "Internal error 2 - transmogrify a fragment (load eax)"; exit.  
If the fragment's variable's type's length is not 4, abort with "Internal error 3 - transmogrify a fragment (load eax)"; exit.  
Attach $8B02 to the fragment's code. \ mov eax,[edx]

To transmogrify a fragment (not):  
Attach $83F001 to the fragment's code. \ xor eax,1

To transmogrify a fragment (prolog):  
Attach $55 to the fragment's code. \ push ebp  
Attach $8BEC to the fragment's code. \ mov ebp,esp  
Put the current routine's local size divided by 4 into a number.  
If the number is not 0, attach $B9 and the number and $6A004975FB to the fragment's code. \ mov ecx,number; loop: push 0; dec ecx; jnz loop  
If the current routine's callback flag is set, attach $535657 to the fragment's code. \ push ebx, esi, edi

To transmogrify a fragment (push address):  
Attach address loading code to the fragment's code given the fragment's variable. \ put address into edx  
Attach $52 to the fragment's code. \ push edx

To transmogrify a fragment (push value):  
Attach address loading code to the fragment's code given the fragment's variable. \ put address into edx  
If the fragment's variable is nil, abort with "Internal error - transmogrify a fragment (push value)"; exit.  
If the fragment's variable's type is nil, abort with "Internal error 2 - transmogrify a fragment (push value)"; exit.  
Put the fragment's variable's type's length into a length.  
If the length is 4, attach $FF32 to the fragment's code; exit. \ push [edx]  
If the length is 2, attach $66FF32 to the fragment's code; exit. \ push word ptr [edx]  
If the length is 1, attach $0FB61252 to the fragment's code; exit. \ movzxb edx,[edx]; push edx  
Abort with "Internal error 3 - transmogrify a fragment (push value)".

To transmogrify a fragment (repeat):  
Find another fragment given the fragment and the loop tag (backwards).  
If the other fragment is nil, abort with "Internal error - transmogrify a fragment (repeat)"; exit.  
Attach $E9 and the other fragment's address to the fragment. \ jmp the destination's address

To transmogrify a fragment (routine address):  
Attach address loading code to the fragment's code given the fragment's variable. \ put address into edx  
If the fragment's routine is nil, abort with "Internal error - transmogrify a fragment (routine address)"; exit.  
Get an address given the fragment's routine.  
Attach $C702 and the address to the fragment's code. \ mov [edx],the address

To transmogrify a fragment (save eax):  
Attach address loading code to the fragment's code given the fragment's variable. \ put address into edx  
If the fragment's variable is nil, abort with "Internal error - transmogrify a fragment (save eax)"; exit.  
If the fragment's variable's type is nil, abort with "Internal error 2 - transmogrify a fragment (save eax)"; exit.  
Put the fragment's variable's type's length into a length.  
If the length is 4, attach $8902 to the fragment's code; exit. \ mov [edx],eax  
If the length is 2, attach $668902 to the fragment's code; exit. \ mov [edx],ax  
If the length is 1, attach $8802 to the fragment's code; exit. \ mov [edx],al  
Abort with "Internal error 3 - transmogrify a fragment (save eax)".

If you're wondering why we only need 18 fragment transmogrifiers when we have 21 fragment types, it's because 3 of those fragment types don't require transmogrification. The END IF, LOOP, and FINALIZE tags are just labels that we stick in there (and address) so we have places to jump to.

247. Some of the transmogrifiers need a little help from this guy...

To attach address loading code to a hex string given a variable: \ loads into edx  
If the variable is nil, attach $C7C200000000 to the hex string; exit. \ mov edx,0  
If the variable's kind is "global", attach $C7C2 and the variable's address to the hex string; exit. \ mov edx,the variable's address  
If the variable's kind is "literal", attach $C7C2 and the variable's address to the hex string; exit. \ mov edx,the variable's address  
If the variable's kind is "local", attach $8D95 and the variable's offset to the hex string; exit. \ lea edx,[ebp+the variable's offset]  
If the variable's kind is "scratch", attach $8B95 and the variable's offset to the hex string; exit. \ mov edx,[ebp+the variable's offset]  
If the variable's kind is not "parameter", abort with "Internal error - attach address loading code to a hex string given a variable"; exit.  
If the variable's by-value flag is set, attach $8D95 and the variable's offset to the hex string; exit. \ lea edx,[ebp+the variable's offset]  
Attach $8B95 and the variable's offset to the hex string. \ mov edx,[ebp+the variable's offset]

248. ...and from these five general-purpose helpers:

To attach an hex string and an address to a fragment: \ call or jump  
Put the address into a number.  
Subtract the fragment's address from the number.  
Subtract the fragment's code's length from the number.  
Subtract the hex string's length from the number.  
Subtract 4 from the number.  
Attach the hex string and the number to the fragment's code.

To attach a hex string and a number and another hex string to a third hex string:  
Append the hex string to the third hex string.  
Convert the number to a fourth hex string.  
Append the fourth hex string to the third hex string.  
Append the other hex string to the third hex string.

To attach a hex string and a number and another number to another hex string:  
Append the hex string to the other hex string.  
Convert the number to a third hex string.  
Append the third hex string to the other hex string.  
Convert the other number to a fourth hex string.  
Append the fourth hex string to the other hex string.

To attach a hex string and a number to another hex string:  
Append the hex string to the other hex string.  
Convert the number to a third hex string.  
Append the third hex string to the other hex string.

To attach a hex string to another hex string: employ append a string to another string.

249. Getting a routine's address isn't as simple as it sounds since each routine's header can be worded in different ways, and since a routine may explicitly employ a different routine to do its work. So this helper is needed to deal with all that:

To get an address given a routine:  
If the routine's address is not 0, put the routine's address into the address; exit.  
Put -1 into the routine's address.  
Find another routine given the routine's employs moniker and the routine index.  
If the other routine is nil, abort with "I can't find the '" then the routine's employs moniker then "' routine you're trying to employ." and the routine's locus; exit.  
If the other routine's address is -1, abort with "Recursive reference in employ clause." and the routine's locus; exit.  
Get the address given the other routine.  
Put the address into the routine's address.

250. We also need these two guys to help us find where to jump to (in various cases):

To find a fragment given another fragment and a tag:  
Put the other fragment into the fragment.  
Loop.  
If the fragment is nil, exit.  
If the fragment's tag is the tag, exit.  
Put the fragment's next into the fragment.  
Repeat.

To find a fragment given another fragment and a tag (backwards):  
Put the other fragment into the fragment.  
Loop.  
If the fragment is nil, exit.  
If the fragment's tag is the tag, exit.  
Put the fragment's previous into the fragment.  
Repeat.  
  
251. And that's it. Transmogrification complete. Run it. In the blue, open the daughter and list her. Then open the listing and jump to the "ROUTINES:" section. This is what you should see there:

ROUTINES:

/routine/abort from/given/with/using [string]/yes/no/no/no//4/12/00419000/  
/variable/parameter/yes/string/string/string/string/00000008/no/1/no////  
/variable/scratch/yes/~S7063//flag/flag/FFFFFFFC/no/1/no////  
/variable/scratch/yes/~S7064//flag/flag/FFFFFFF8/no/1/no////  
/variable/scratch/yes/~S7065//abort message/abort message/FFFFFFF4/no/1/no////  
/fragment/prolog/////00000000/00419000/558BECB9030000006A004975FB/  
/fragment/loop/////00000000/0041900D//  
/fragment/load address/compiler/~S7063///00000000/0041900D/C7C2402040008995FCFFFFFF/  
/fragment/increment/~S7063////00000014/00419019/8185FCFFFFFF14000000/  
/fragment/push address/~S7063////00000000/00419023/8B95FCFFFFFF52/  
/fragment/call internal///[flag] is/are set//00000000/0041902A/E899160300/  
/fragment/jump false/////00000000/0041902F/83F8000F8405000000/  
/fragment/exit/////00000000/00419038/E94B000000/  
/fragment/end if/////00000000/0041903D//  
/fragment/load address/compiler/~S7064///00000000/0041903D/C7C2402040008995F8FFFFFF/  
/fragment/increment/~S7064////00000014/00419049/8185F8FFFFFF14000000/  
/fragment/push address/~S7064////00000000/00419053/8B95F8FFFFFF52/  
/fragment/call internal///set [flag]//00000000/0041905A/E8117B0500/  
/fragment/load address/compiler/~S7065///00000000/0041905F/C7C2402040008995F4FFFFFF/  
/fragment/increment/~S7065////00000018/0041906B/8185F4FFFFFF18000000/  
/fragment/push address/~S7065////00000000/00419075/8B95F4FFFFFF52/  
/fragment/push address/string////00000000/0041907C/8B950800000052/  
/fragment/call internal///put [string] in/into/to [string]//00000000/00419083/E8F8130500/  
/fragment/finalize/////00000000/00419088//  
/fragment/epilog/////00000000/00419088/8BE55DC204000000/

Look at all that lovely machine code on the end of each fragment line! (Except for the LOOP, END IF, and FINALIZE lines, of course).

Next time, we cram everything we've got into a Portable Executable.

---

Coming down the home stretch now. Time to cram everything into a Portable Executable file and finish up.

252. Let's begin by grabbing the three ugly types we need for Windows:

A dos header is a record with  
A wyrd called signature,  
A wyrd called numberofbytesonlastpage,  
A wyrd called numberofpages,  
A wyrd called relocuss,  
A wyrd called sizeofheaderinparagraphs,  
A wyrd called minimumextraparagraphs,  
A wyrd called maximumextraparagraphs,  
A wyrd called initialrelativess,  
A wyrd called initialsp,  
A wyrd called initialchecksum,  
A wyrd called initialip,  
A wyrd called initialirelativecs,  
A wyrd called addressofrelocationtableinfile,  
A wyrd called overlynumber,  
8 bytes called resveredbytes,  
A wyrd called oemidentifier,  
A wyrd called oeminfo,  
20 bytes called reservedwords,    
A number called addressofpeheaderinfile.

A pe header is a record with  
A number called signature,  
A wyrd called machinetype,  
A wyrd called numberofsections,  
A number called timestamp,  
A number called pointertosymboltable,  
A number called numberofsymbols,  
A wyrd called sizeofoptionalheaderinbytes,  
A wyrd called characteristics,  
\ optional header  
A wyrd called magicnumber,  
A byte called majorlinkerversion,  
A byte called minorlinkerversion,  
A number called sizeofcodeinfile,  
A number called sizeofinitializeddatainfile,  
A number called sizeofuninitializeddatainfile,  
A number called addressofentrypointinmemory,  
A number called addressofcodeinmemory,  
A number called addressofinitializeddatainmemory,  
A number called imagebase,  
A number called memoryalignment,  
A number called filealignment,  
A wyrd called osmajorversion,  
A wyrd called osminorversion,  
A wyrd called usermajorversion,  
A wyrd called userminorversion,  
A wyrd called subsystemmajorversion,  
A wyrd called subsystemminorversion,  
A number called reserved,  
A number called sizeofimageinmemory,  
A number called sizeofallheadersinfile,  
A number called checksum,  
A wyrd called subsystem,  
A wyrd called dllcharacteristics,  
A number called maxstack,  
A number called minstack,  
A number called maxheap,  
A number called minheap,  
A number called loaderflags,  
\ directories (16 address and size pairs)  
A number called numberofdirectories,  
8 bytes called directory0,  
A number called imagedirectoryentryimportaddress,  
A number called imagedirectoryentryimportsize,  
112 bytes called directories2-15,  
A pe section header called idata section,  
A pe section header called data section,  
A pe section header called code section.

A pe section header is a record with  
8 bytes called name,  
A number called sizeinbytes,  
A number called addressinmemory,  
A number called sizeinfile,  
A number called addressinfile,  
12 bytes called reserved,  
A number called characteristics.

253. Then let's delete the abort line altogether, and activate the the three remaining steps:

To compile a directory:  
Compile the directory (start).  
Compile the directory (load the source files).  
Compile the directory (scan the source files).  
Compile the directory (resolve the types).  
Compile the directory (resolve the globals).  
Compile the directory (compile the headers of the routines).  
Compile the directory (calculate lengths and offsets of types).  
Compile the directory (add the built-in memory routines).  
Compile the directory (index the routines for utility use).  
Compile the directory (compile the bodies of the routines).  
Compile the directory (add and compile the built-in startup routine).  
Compile the directory (offset parameters and variables).  
Compile the directory (address).  
Compile the directory (transmogrify).  
Compile the directory (link).  
Compile the directory (write the exe).  
Compile the directory (stop).

254. Then let's add this guy back in:

To compile a directory (link):  
If the compiler's abort flag is set, exit.  
Show status "Linking...".  
Start the linking timer.  
Link.  
Stop the linking timer.

255. And this guy:

To link:  
Round up borough sizes.  
Initialize a dos header.  
Initialize a pe header.  
Fill the compiler's exe with the null byte given the compiler's exe size.  
Blurt the dos header into the compiler's exe.  
Blurt the pe header into the compiler's exe.  
Blurt the imports into the compiler's exe.  
Blurt the globals into the compiler's exe.  
Blurt the literals into the compiler's exe.  
Blurt the routines into the compiler's exe.

256. Now some helpers, starting with this routine:

To round up borough sizes:  
Put the import borough's length into the import borough's size.  
Round the import borough's size up to the nearest multiple of 4096.  
Put the data borough's length into the data borough's size.  
Round the data borough's size up to the nearest multiple of 4096.  
Put the code borough's length into the code borough's size.  
Round the code borough's size up to the nearest multiple of 4096.

257. And let's try not to barf as we add these 9 guys back in:

To initialize a dos header:  
Put 23117 [$5A4D] into the dos header's signature.  
Put 64 [$0040] into the dos header's addressofrelocationtableinfile.  
Put 256 [$00000100] into the dos header's addressofpeheaderinfile.

To initialize a pe header:  
Initialize the pe header (standard header).  
Initialize the pe header (optional header).  
Initialize the pe header (version information).  
Initialize the pe header (directories).  
Initialize the pe header (import section).  
Initialize the pe header (data section).  
Initialize the pe header (code section).

To initialize a pe header (code section):  
Copy bytes from "code  "'s first to the pe header's code section's name's whereabouts for 6.  
Put the code borough's length into the pe header's code section's sizeinbytes.  
Put the code borough's base into the pe header's code section's addressinmemory.  
Put the code borough's length into the pe header's code section's sizeinfile.  
Put the code borough's base into the pe header's code section's addressinfile.  
Put -536870880 [$E0000020] into the pe header's code section's characteristics. \ executable, readable, writable, code object

To initialize a pe header (data section):  
Copy bytes from "data  "'s first to the pe header's data section's name's whereabouts for 6.  
Put the data borough's length into the pe header's data section's sizeinbytes.  
Put the data borough's base into the pe header's data section's addressinmemory.  
Put the data borough's length into the pe header's data section's sizeinfile.  
Put the data borough's base into the pe header's data section's addressinfile.  
Put -1073741760 [$C0000040] into the pe header's data section's characteristics. \ initialized, readable, writable

To initialize a pe header (directories):  
Put 16 [$00000010] into the pe header's numberofdirectories.  
Put the import borough's base into the pe header's imagedirectoryentryimportaddress.  
Put the import borough's length into the pe header's imagedirectoryentryimportsize.

To initialize a pe header (import section):  
Copy bytes from "idata "'s first to the pe header's idata section's name's whereabouts for 6.  
Put the import borough's length into the pe header's idata section's sizeinbytes.  
Put the import borough's base into the pe header's idata section's addressinmemory.  
Put the import borough's length into the pe header's idata section's sizeinfile.  
Put the import borough's base into the pe header's idata section's addressinfile.  
Put -1073741760 [$C0000040] into the pe header's idata section's characteristics. \ initialized, readable, writable

To initialize a pe header (optional header):  
Put 267 [$010B] into the pe header's magicnumber.  
Put the code borough's length into the pe header's sizeofcodeinfile.  
Put the data borough's length into the pe header's sizeofinitializeddatainfile.  
Put 0 into the pe header's sizeofuninitializeddatainfile.  
Find a routine given "~initialize before run and run and finalize after run" and the routine index.  
If the routine is nil, abort with "Internal error: could not find the startup routine"; exit.  
Put the routine's address minus the image base into the pe header's addressofentrypointinmemory.  
Put the code borough's base into the pe header's addressofcodeinmemory.  
Put the data borough's base into the pe header's addressofinitializeddatainmemory.  
Put the image base into the pe header's imagebase.  
Put 4096 [$00001000] into the pe header's memoryalignment.  
Put 4096 [$00001000] into the pe header's filealignment.  
Put 0 into the pe header's reserved.  
Put the compiler's exe size into the pe header's sizeofimageinmemory.  
Put 4096 [section base] into the pe header's sizeofallheadersinfile.  
Put 0 into the pe header's checksum.  
Put 0 into the pe header's dllcharacteristics.  
Put 1048576 [$00100000] into the pe header's maxstack.  
Put 16384 [$00004000] into the pe header's minstack.  
Put 1048576 [$00100000] into the pe header's maxheap.  
Put 16384 [$00004000] into the pe header's minheap.  
Put 0 into the pe header's loaderflags.

To initialize a pe header (standard header):  
Put 17744 [$00004550] into the pe header's signature.  
Put 332 [$014C] into the pe header's machinetype.  
Put 3 [$0003] into the pe header's numberofsections.  
Put 0 into the pe header's timestamp.  
Put 0 into the pe header's pointertosymboltable.  
Put 0 into the pe header's numberofsymbols.  
Put 224 [$00E0] into the pe header's sizeofoptionalheaderinbytes.  
Put 33166 [$818E] into the pe header's characteristics.

To initialize a pe header (version information):  
Put 0 into the pe header's majorlinkerversion.  
Put 0 into the pe header's minorlinkerversion.  
Put 4 [$0004] into the pe header's osmajorversion.  
Put 0 [$0000] into the pe header's osminorversion.  
Put 0 into the pe header's usermajorversion.  
Put 0 into the pe header's userminorversion.  
Put 4 [$0004] into the pe header's subsystemmajorversion.  
Put 0 [$0000] into the pe header's subsystemminorversion.  
Put 2 [$0002] into the pe header's subsystem.

258. We'll also be needing this little function-style routine:

To put the compiler's exe size into a size:  
Put 4096 [section base] into the size.  
Add the import borough's size to the size.  
Add the data borough's size to the size.  
Add the code borough's size to the size.

259. And we'll be needing all 15 of our "blurt" routines, that mostly just copy bytes (quickly) from our various lists to the right spots in our pre-allocated executable buffer in memory.

To blurt bytes given a pointer and a length and a buffer and an offset:  
Put the buffer's first plus the offset into another pointer.  
Copy bytes from the pointer to the other pointer for the length.

To blurt a dos header into a buffer:  
Blurt bytes given the dos header's whereabouts and the dos header's magnitude and the buffer and 0.

To blurt some entries into a buffer:  
Get an entry from the entries.  
If the entry is nil, exit.  
Blurt the entry into the buffer.  
Repeat.

To blurt an entry into a buffer:  
Blurt bytes given the entry's name address's whereabouts and 4 and the buffer and the entry's thunk address.  
Put the entry's name address plus 2 into an address.  
Blurt the entry's name into the buffer at the address.

To blurt a fragment into a buffer:  
If the fragment's code is blank, exit.  
Put the fragment's address minus the image base into an address.  
Blurt the fragment's code into the buffer at the address.

To blurt some fragments into a buffer:  
Get a fragment from the fragments.  
If the fragment is nil, exit.  
Blurt the fragment into the buffer.  
Repeat.

To blurt an import into a buffer:  
Blurt bytes given the import's import header's whereabouts and the import's import header's magnitude and the buffer and the import's header address.  
Blurt the import's name into the buffer at the import's name address.  
Blurt the import's entries into the buffer.

To blurt some imports into a buffer:  
Get an import from the imports.  
If the import is nil, exit.  
Blurt the import into the buffer.  
Repeat.

To blurt a pe header into a buffer:  
Blurt bytes given the pe header's whereabouts and the pe header's magnitude and the buffer and 256.

To blurt a routine into a buffer:  
If the routine's compiled flag is not set, exit.  
Blurt the routine's fragments into the buffer.

To blurt some routines into a buffer:  
Get a routine from the routines.  
If the routine is nil, exit.  
Blurt the routine into the buffer.  
Repeat.

To blurt a string into a buffer at an address:  
Blurt bytes given the string's first and the string's length and the buffer and the address.

To blurt a variable into a buffer:  
If the variable's compiled flag is not set, exit.  
If the variable's type can be reduced to "string", blurt the variable into the buffer (as a string); exit.  
Put the variable's address minus the image base into an address.  
Blurt the variable's data into the buffer at the address.

To blurt a variable into a buffer (as a string):  
Put the variable's address minus the image base into an address.  
Put the variable's data's length into a length.  
Put the address into another address.  
Add a prototype string's magnitude to the other address.  
Add the image base to the other address.  
Put 0 into the prototype string's first.  
Put -1 into the prototype string's last.  
If the length is not 0, put the other address into the prototype string's first.  
If the length is not 0, put the prototype string's first plus the length minus 1 into the prototype string's last.  
Blurt bytes given the prototype string's whereabouts and the prototype string's magnitude and the buffer and the address.  
Add the prototype string's magnitude to the address.  
Blurt the variable's data into the buffer at the address.

To blurt some variables into a buffer:  
Get a variable from the variables.  
If the variable is nil, exit.  
Blurt the variable into the buffer.  
Repeat.

At this point we've got a complete executable sitting in our buffer in memory, and we're done with the "link" step.

260. On to the next step; add this guy back in:

To compile a directory (write the exe):  
If the compiler's abort flag is set, exit.  
Show status "Writing...".  
Start the writing timer.  
Write the compiler's exe to the compiler's exe path.  
If the i/o error is not blank, abort with the i/o error.  
Stop the writing timer.

261. And finally, this guy:

To compile a directory (stop):  
Stop the compiler's timer.  
Show status "".

262. almost there; just one little clean-up: At the top of the file there are two comments that got double-commented when we commented out the whole file way back when. Change those double-backslashes to just single ones.

TA DA! We did it! A complete Plain English compiler written entirely in Plain English.

263. Run it. In the blue, open the SON's directory up, and copy his now-complete compiler file to the clipboard (select the file, then hit CTRL-C). Then open the daughter's directory and paste (CTRL-V) to overwrite the genetically defective compiler file she's got now, so she'll also be able to reproduce in the future. Then open up and find this line in the daughter's Desktop file...

Put the lightest sky color into the desktop color.

...and change it to read:

Put the pink color into the desktop color.

Run it. Congratulations! It's a girl! And you now have three healthy generations of the CAL running. ALT-TAB to see them all in a bunch.

If you want to see the third-generation Portable Executable we just generated, open the daughter's directory, find the ".exe" in there, and open it (as a dump). There she is (a real "binary beauty") starting with the lovely DOS header:

00000000  4D 5A 00 00 00 00 00 00 00 00 00 00 00 00 00 00  MZ..............  
00000010  00 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ........@.......  
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  
00000030  00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00  ................  
00000040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  
00000050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  
00000060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  
00000070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  
00000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  
00000090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  
000000A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  
000000B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  
000000C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  
000000D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  
000000E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................  
000000F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

And that's all I have to say about that. Questions?

Gerry