EXHIBIT 12: LEWIS IN VIEW OF DYER AND FURTHER IN VIEW OF PROEBSTING	Clarity MCode: A Retargetable Intermediate Representation for Compilation Brian T. Lewis at L. Peter Deutsch and Theodore C. Goldstein ACM, IR '95, 1/95, San Francisco, California, USA (1995) (''Lewis'')	<u>Java Decompilers Compared</u> Dave Dyer JavaWorld.com (July 1, 1997) ("Dyer")	 Krakatoa: Decompilation in Java (Does Bytecode Reveal Source?) Todd A. Proebsting and Scott A. Watterson Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems, Portland, Oregon (June 1997) ("Proebsting") 	ILewis in view of Dyer and further in view of ProebstingmLewis discloses a method in a data processing system, i.e. the "Clarity MCode compilation system," for statically initializing an array. For example, Lewis discloses that the MCode instruction set includes instructions for statically initializing an array, e.g., "AllocArray.""This subsection presents an overview of the Clarity MCode compilation system; more details are given in Section 3. There are two major parts to the MCode compilation system: a program development-time part and a runtime part. The development-time portion consists of an MCode generator for the Clarity language and a Linkable MCode converter; see Figure 1. The MCode generator reads semantically decorated Clarity ASTs stored in the Clarity programming environment's database and produces platform-independent MCode." Lewis at 120.
EXHIBIT 12	<u>Clarity MC</u> Bri A		Krakı	U.S. Patent No. 6,061,520 – Claim 1 1. A method in a data processing system for statically initializing an array, comprising the steps of:

4. 4. 4.		
	AllocArray(array type index)	iđ 5wi
<u>حر</u>	Arrayindex(array type index)	swike
	AmayLength(array type index)	ផ្លំពន់រូ
		kse()
	InvokeOuter(method type index)	witch
	invokeDelegated(method type index) Widen(base interface number)	Ĩ
2	Narrow(target type index)	ದ ಬಿತ್ತು)
	Figure 4: The MCode instruction set	set
Te	Lewis at 123 Fig. 4 (excerpt).	
Dy Cla	<i>Dyer</i> discloses a review of three Java Decompilers. These are programs that convert Java class files into Java source code, effectively reverse engineering compiled code to figure out how the underlying code works. <i>Dyer</i> discloses, in most relevant part, an example of the	tt convert Java ode to figure out xample of the
coo dis to 1 tes	code that would be used to implement a static initialization of an array. <i>Dyer</i> not only discloses the concept of the static initialization of an array by way of an instruction provided to the Java virtual machine; <i>Dyer</i> also provides examples of actual code to achieve this result.	er not only ruction provided chieve this
u/T d'T	The decompiler of <i>Dyer</i> inherently performs a symbolic execution of the compiled bytecode. The functionality of such a decompiler, including the symbolic execution that decompilers	mpiled bytecode. at decompilers
	use to identify the target of compiled bytecode for the recovery of the underlying Java source code, is disclosed in detail in <i>Proebsting</i> . The combination of the <i>Proebsting</i>	lying Java roebsting
dis res ob	disclosure, which explains the symbolic execution technique of a decompiler, with the actual results of a decompiler as applied to a static initializer for an array, as shown by <i>Dyer</i> , render obvious the static initialization of an array as in claim 1.	r, with the actual 1 by <i>Dyer</i> , render
compiling source code containing the <i>Le</i> array with static values to generate a cla	<i>Lewis</i> discloses compiling source code containing the array with static values to generate a class file with a clinit method containing byte codes to statically initialize the array to the	s to generate a e array to the
	static values. For example, <i>Lewis</i> discloses compiling source code written in the Clarity C++ programming language into MCode object files, and that the MCode instruction set	in the Clarity struction set
initialize the array to the static values; inc	includes instructions for statically initializing arrays.	

U.S. Patent No. 6,061,520 - Claim 1	Lewis in view of Dyer and further in view of Proebsting
	"The <i>Clarity C</i> ++ programming language is a dialect of C++ being developed in Sun Microsystems Laboratories. Clarity shares many features with C++ but is less complex and has a more consistent syntax and simpler semantics without loss in expressiveness Clarity is intended to be a wide-spectrum language suitable for both systems and application programming, particularly of distributed software.
	To support the compilation of Clarity, we have developed a high-level, machine-independent intermediate representation that we call <i>MCode</i> (for "middle code"). We use MCode to compile Clarity programs at execution time (i.e., on-the-fly) into SPARC code for the Solaris operating system. This code generator is designed to be largely machine independent: besides the SPARC code generator, an Intel x86 version is being developed." <i>Lewis</i> at 119 (footnote omitted).
	"This subsection presents an overview of the Clarity MCode compilation system; more details are given in Section 3. There are two major parts to the MCode compilation system: a program development-time part and a runtime part. The development-time portion consists of an MCode generator for the Clarity language and a Linkable MCode converter; see Figure 1. The MCode generator reads semantically decorated Clarity ASTs stored in the Clarity programming environment's database and produces platform-independent MCode. The Linkable MCode converter then wraps a compact encoding of the MCode into a standard object file. Linkable MCode object files are then combined by standard linkers with other object files to produce executables and shared libraries."
	Editor Figure 1: The development-time portion of the Clarity MC ode system
	<i>Lewis</i> at 120-21, Fig. 1.

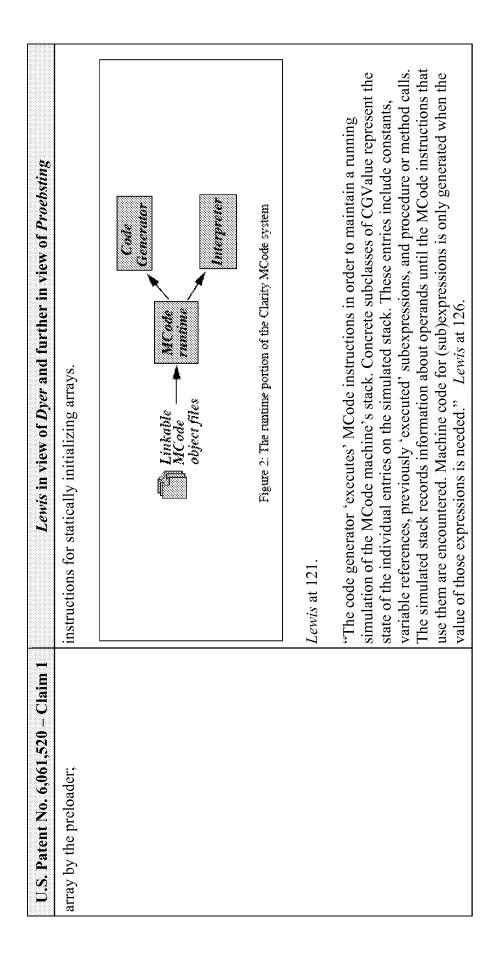
Э

Lewis in view of Dyer and further in view of Proebsting	"Linkable MCode object files contain a machine-independent <i>pickle</i> of an MCode code unit. This is a compact, platform-independent encoding of the MCode information into a sequence of bytes. This pickle can later be internalized or unpickled to reconstruct the original MCode. The MCode for each procedure is pickled separately to support procedure-at-a-time processing. The current encoding is not especially compact although the Linkable MCode object files are still smaller than object files containing machine code. We intend to replace the current pickle format with a more compact one.	Linkable MCode object files are platform-standard object files that are processed in the usual way by the platform's standard linker. This means they need to include platform-dependent definitions of global variables and procedures, and descriptions of referenced symbols. We currently encode ("mangle") symbol names in order to ensure that the resulting executables or shared libraries are type-safe with respect to the Clarity language. Eventually, this type-safety will be checked by a Clarity <i>prelinker</i> .	Besides symbol definitions and references, our Solaris Linkable MCode object files also contain a few machine language instructions for each procedure's entry code. This entry code allows C code allows C code to call the MCode procedure. On the SPARC, this entry code consists primarily of a three instruction "trampoline" that redirects the call to the appropriate target procedure chosen by the interpret/compile strategy module in the MCode runtime. The SPARC entry code also has three words used when atomically updating the trampoline. Despite this platform-specific information, the contents of a Linkable MCode file are mostly platform-independent. We currently execute the Linkable MCode converter during program development, before a program is distributed. It could also be executed when the program installed on a particular platform." <i>Lewis</i> at 125-26.
U.S. Patent No. 6,061,520 - Claim 1			

U.S. Patent No. 6,061,520 - Claim 1	Lewis in view of Dyer and further in view of Proebsting	rther in view of <i>Proebsting</i>
	AllocArray(array type index)	BeginSwitch(end switch tag)
	ArrayIndex(array type index)	EndSwitch(end switch tag)
	ArrayLength(array type index)	BeginExprCase(int)
		BeginDefauftCase()
	InvokeOuter(method type index)	EndCase(end switch tag)
	InvokeDelegated(method type index)	
	Widen(base interface number)	DoBreak(end tag)
	Narrow(target type index)	DoContinue(end tag)
	Figur	Figure 4: The MCode instruction set
	Lewis at 123 Fig. 4 (excerpt).	
	Figures 5 and 6 give an example of generated MCode. The Clarity method <i>startup</i> in Figure 5 wordness the MCode instructions shown in part in Figure 6."	MCode. The Clarity method <i>startup</i> in <i>m</i> in wart in Figure 6."
		111 III puit 111 1 15ul ~ 0.

U.S. Patent No. 6,061,520 - Claim 1	Lewis in view of Dyer and further in view of Proebsting
	ThreadedSimulation: module
	<pre>* work_mutex: Threads:NMutex; // protected by work_mutex work_per_work: int; // protected by work_mutex extra_work: int; // protected by work_mutex</pre>
	Worker: type = interface inherits 7 threads::Thread ϑ ; <i>H</i> an unusual interface: no methods beyond Thread::startup and the other 7 hread methods
	Worklmpk: type = implementation of Worker
	<pre>implement startup: method (our_workers: in int)</pre>
	$x \in xtra_work > U \in xtra_work += 1; extra_work -= 3; $
	<pre>detegated_to_workers: ini = (our_workers - 1); if (detegated_to_workers > 0) {</pre>
	right_sibling = new Workerimp(right_workers); } do_work(our_work);
	// the following declarations are private to the WorkerImpl implementation do_work: method {work_to_do: in int} { /* elided */ };
	อนr_work: int = 0; left_workers: int = 0; right_workers: int = 0; left_sibiling: Worker; // ieft delegatee; manages "eft_workers" workers right_workers" workers
	Figure 5: Part of the Clarity version of the μC^{++} test program
	Lewis at 124, Fig. 5.

Lewis in view of Dyer and further in view of Proebsting	<i>Lewis</i> discloses receiving the class file, i.e. the "MCode object files," into a preloader, i.e., the "Code Generator."	Linkable Mt ode Code Object files	Figure 2: The runtime portion of the Clarity MCode system	"The runtime component of the MCode system is illustrated in Figure 2. The MCode runtime in an MCode-containing executable internalizes the MCode for each procedure as needed, when the procedure is first called. It also implements a interpret/code generate policy separately for each MCode procedure. This policy chooses for each procedure whether to interpret it, generate code, or interpret then later generate code, or generate better code. The code generate code interpret it generate code interpret then later generate code, or generate better the SunPRO C compiler at the default -O2 optimization level. A port of the code generator to the x86 is underway. The MCode interpreter interoperates with all SPARC ABI code. Like the compiler, it is reentrant and supports multithreaded programs. It also does extensive checking during program execution, which makes it especially useful for uncovering errors in Clarity programs that are otherwise difficult to detect. The interpreter will also be used by the Clarity debugger that we are developing to evaluate Clarity statements and expressions." <i>Lewis</i> at 120-21.	<i>Lewis</i> discloses simulating execution of the byte codes of the clinit method against a memory without executing the byte codes to identify the static initialization of the array by the preloader. For example, <i>Lewis</i> discloses simulating execution of the MCode instructions in the MCode object files, and that the MCode instruction set includes
U.S. Patent No. 6,061,520 - Claim 1	receiving the class file into a preloader;				simulating execution of the byte codes of the clinit method against a memory without executing the byte codes to identify the static initialization of the



U.S. Patent No. 6,061,520 - Claim 1	Lewis in view of Dyer and further in view of Proebsting	bsting
	AllocArray(array type index) Begin	BeginSwitch(end switch tag)
	ArrayIndex(array type index) EndS	EndSwitch(end switch tag)
	ArrayLength(array type index) Begin	BeginExprCase(int)
	Begin	BeginDefauftCase()
	InvokeOuter(method type index) EndC	EndCase(end switch tag)
	InvokeDelegated(method type index)	
	Widen(base interface number) DoBn	DoBreak(end tag)
	Narrow(target type index) DoCo	DoContinue(end tag)
	Figure 4: The MCode instruction set Lewis at 123 Fig. 4 (excerpt).	nuction set
	To the extent <i>Lewis</i> does not explicitly disclose this limitation, one of ordinary skill would	ordinary skill would
	Dyer and Proebsting together show that the simulation of byte codes against a memory to identify underlying code, such as the static initialization of an array, was commonly known	against a memory to vas commonly known
	to those of ordinary skill in the art at the time of the invention. For example, the decompiler technology disclosed by <i>Proebsting</i> discloses a technique for decompilation of Taya byte code that simulates the Taya with all machine's stack:	example, the for decompilation of
	Java dyle couc ulai shihuates ule Java Villuai hiachine S siack.	
	"This paper presents our technique for automatically decompiling Java bytecode into Java source." <i>Proebsting</i> at Abstract.	a bytecode into Java
	"Decompilation transforms a low-level language into a high-level language.	guage. The Java
	Virtual machine (JVM) specifies a low-level bytecode language for a stack-based machine. This language defines 203 operators, with most of the control flow specified by simple	stack-based machine. ecified by simple
	explicit transfers and labels. Compiling a Java class yields a <i>class file</i> that contains type	le that contains type
	from the class files for object linking Decompilation systems can exploit this type of	an exploit this type of
	information and well-behaved property to recover Java source code from the class file.	om the class file.

U.S. Patent No. 6,061,520 - Claim 1	Lewis in view of Dyer and further in view of Proebsting
	We present a technique for transforming low-level Java bytecode into legal Java source code." <i>Proebsting</i> § 1.
	"We have implemented a prototype Java decompiler, Krakatoa, in Java." Proebsting § 5.
	"Krakatoa uses a stack simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations." <i>Proebsting</i> § 1.
	"Symbolic execution of the bytecode creates the corresponding Java source expressions Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." <i>Proebsting</i> § 2.
	Thus it would have been obvious to one of ordinary skill in the art to combine the methods of <i>Lewis</i> with the simulation (symbolic execution) of bytecode against a memory, as disclosed by <i>Proebsting</i> . And, even more specifically, the artisan would look to known decompilers, such as those disclosed in <i>Dyer</i> . <i>Dyer</i> reviews three popular decompilers that were available before the '520 patent was filed: DejaVu, Mocha, and WingDis. <i>See Dyer</i> at 1. In that review, <i>Dyer</i> analyzes the decompilation of code, including the decompilation of static initializers:
	"Mocha transformed a static initializer into an elegant, but illegal, construction:
	<pre>public ConsoleWindow(String string, int il) {</pre>
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these

U.S. Patent No. 6,061,520 - Claim 1	Lewis in view of Dyer and further in view of Proebsting
	initializers are actually implemented by inline code inside constructors, generated by the compiler.
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." <i>Dyer</i> at 3.
	The fact that some of $Dyer$'s decompiled code may be considered "illegal" should not detract from the fact that the decompilation of static array initialization byte code into a single expression or instruction was a well-known technique at the time the '520 patent was filed a technique used to identify the operation of the byte code in order to generate
	equivalent nign-level language expressions. In fact, the '220 patent discloses creating a constant pool entry that is not a standard Java virtual machine construct. Thus, one of ordinary skill in the art would have considered this constant pool construct (i.e., CONSTANT Array (500, 570 patent at 8.54.0.13)) as an "illegal? Java constant pool construct entry.
	The reason it wouldn't have mattered is that the '520 patent states that "the virtual machine 222 is modified to recognize the static initialization directives of the preloader." '520 patent at 4:46-48. Because the exact form of modification is not disclosed, it must be within the ordinary artisan's skill set to perform the correct modification to allow the virtual
	machine to recognize an "illegal" instruction such as the new CONSTANT_Array type. Further during Examination claim terms are to be given their broadest reasonable.
	interpretation (<i>see</i> MPEP § 2111), and there are no qualifiers in the claim language as to the generated instruction. Again, because the '520 patent assumes that one of ordinary skill in the art would modify the virtual machine to recognize the generated instruction, the exact
	format (whether syntactically correct or not) is not an issue. And in any case, <i>Dyer</i> discloses that in contrast to the Mocha decompiler's "illegal" code the WingDis decompiler produced "syntactically correct code." <i>Dyer</i> at 3.
storing into an output file an instruction requesting the static initialization of the array; and	<i>Lewis</i> discloses storing into an output file, i.e., the "Linkable MCode object file," an instruction requesting the static initialization of the array.

Lewis in view of Dyer and further in view of Proebsting	"This subsection presents an overview of the Clarity MCode compilation system; more details are given in Section 3. There are two major parts to the MCode compilation system: a program development-time part and a runtime part. The development-time portion consists of an MCode generator for the Clarity language and a Linkable MCode converter; see Figure 1. The MCode generator reads semantically decorated Clarity ASTs stored in the Clarity programming environment's database and produces platform-independent MCode. The Linkable MCode converter then wraps a compact encoding of the MCode into a standard object file. Linkable MCode object files are then combined by standard linkers with other object files to produce executables and shared libraries."	Editor Figure 1: The development-time portion of the Clarity MC ode object files	<i>Lewis</i> at 120-21, Fig. 1.	"Linkable MCode object files contain a machine-independent <i>pickle</i> of an MCode code unit. This is a compact, platform-independent encoding of the MCode information into a sequence of bytes. This pickle can later be internalized or unpickled to reconstruct the original MCode. The MCode for each procedure is pickled separately to support procedure-at-a-time processing. The current encoding is not especially compact although the Linkable MCode object files are still smaller than object files containing machine code. We intend to replace the current pickle format with a more compact one.	Linkable MCode object files are platform-standard object files that are processed in the usual way by the platform's standard linker. This means they need to include platform-dependent definitions of global variables and procedures, and descriptions of referenced symbols. We currently encode ('mangle') symbol names in order to ensure that the resulting executables or shared libraries are type-safe with respect to the Clarity
U.S. Patent No. 6,061,520 - Claim 1					

U.S. Patent No. 6,061,520 - Claim 1	Lewis in view of Dyer and further in view of Proebsting	ı view of <i>Proebsting</i>
	language. Eventually, this type-safety will be checked by a Clarity <i>prelinker</i> . Besides symbol definitions and references, our Solaris Linkable MCode object files also contain a few machine language instructions for each procedure's entry code. This entry code allows C code to call the MCode procedure. On the SPARC, this entry code consists primarily of a	by a Clarity <i>prelinker</i> . Besides MCode object files also contain a s entry code. This entry code allows his entry code consists primarily of a
	three instruction 'trampoline' that redirects the call to the appropriate target procedure chosen by the interpret/compile strategy module in the MCode runtime. The SPARC entry code also has three words used when atomically updating the trampoline. Despite this platform-specific information, the contents of a Linkable MCode file are mostly	he appropriate target procedure MCode runtime. The SPARC entry ng the trampoline. Despite this le MCode file are mostly
	platform-independent. The Linkable MCode converter itself is also mostly platform-independent. We currently execute the Linkable MCode converter during program development, before a program is distributed. It could also be executed when the program is	itself is also mostly ole MCode converter during program also be executed when the program is
	IIISTALICU ULI A PALILCULAI PLAUOLIII. LEWIS AL 123-20.	
	AllocAnay(anay type index) Amawindev(arrev tyne index)	BeginSwitch(end swi FordSwitchiend switc
	to approved state of the state in the state	Recipitor and the
	fyerset addit framework a first	BeginDefaultCase()
	invokeOuter(method type index)	EndCase(end switch
	InvokeDelegated(method type index) Widen(base interface number)	DoBreak(end tag)
	Narrow(target type index)	DoContinue(end tag)
	Figure 4: Th	Figure 4: The MCode instruction set
	Lewis at 123 Fig. 4 (excerpt).	
	To the extent <i>Lewis</i> does not explicitly disclose this limitation, one of ordinary skill would combine <i>Lewis</i> with <i>Proebsting</i> and <i>Dyer</i> to arrive at the claimed feature. Specifically, <i>Proebsting</i> discloses that the Krakatoa decompiler writes the source code (which would	itation, one of ordinary skill would ne claimed feature. Specifically, es the source code (which would
	necessarily include any static array initialization instructions) to an output file:	ctions) to an output file:
	"We have implemented a prototype Java decompiler, Krakatoa, in Java.	crakatoa, in Java. We have run

U.S. Patent No. 6,061,520 - Claim 1	Lewis in view of Dyer and further in view of Proebsting
	Krakatoa on a number of class file, including some to which we had no source code access. We examined the output of Krakatoa by hand, and Krakatoa appears to recover high-level constructs very well." <i>Proebsting</i> § 5.
	"Krakatoa is very efficient at reproducing readable Java source from Java bytecode." Proebsting § 6.
	<i>Dyer</i> discloses that the output of a decompiler (such as <i>Proebsting's</i> Krakatoa decompiler) may be an instruction that "request[s] the static initialization of the array," as recited in the claim:
	"Mocha transformed a static initializer into an elegant, but illegal, construction:
	<pre>public ConsoleWindow(String string, int il) {</pre>
	<pre>dead = false; styles = { "Plain", "Bold", "Italic" }; sizes = { "8", "9", "10", "12", "14", "16", "18", "24" }; </pre>
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." <i>Dyer</i> at 3.
	The fact that some of $Dyer$'s decompiled code may be considered "illegal" should not detract from the fact that the decompilation of static array initialization byte code into a single expression or instruction was a well-known technique at the time the '520 patent was filed a technique used to identify the operation of the byte code in order to generate

U.S. Patent No. 6,061,520 - Claim 1	Lewis in view of Dyer and further in view of Proebsting
	equivalent high-level language expressions. In fact, the '520 patent discloses creating a constant pool entry that is not a standard Java virtual machine construct. Thus, one of ordinary skill in the art would have considered this constant pool construct (i.e., CONSTANT_Array (<i>see</i> '520 patent at 8:54-9:13)) as an "illegal" Java constant pool entry. The reason it wouldn't have mattered is that the '520 patent states that "the virtual machine 222 is modified to recognize the static initialization directives of the preloader." '520
	Further, during Examination claim terms are to be given the broadest removed. It must be
	interpretation (<i>see</i> MPEP § 2111), and there are no qualifiers in the claim language as to the generated instruction. Again, because the '520 patent assumes that one of ordinary skill in the art would modify the virtual machine to recognize the generated instruction, the exact format (whether syntactically correct or not) is not an issue. And in any case, <i>Dyer</i> discloses that in contrast to the Mocha decompiler's "illegal" code the WingDis decompiler produced "syntactically correct code." <i>Dyer</i> at 3.
	Additionally, the recitation of this method step in Claim 1 of the '520 patent is not limited with respect to the order in which it is performed relative to the other steps of Claim 1. Thus, the combined systems of <i>Lewis, Proebsting</i> , and <i>Dyer</i> may perform this step (i.e., "storing into an output file an instruction") while it generates a class file, i.e., after it performs the step of "compiling source code" and before the class file is received by the preloader. Or it may perform this step after the class file is received by the preloader. It is step after the class file is received by the preloader. Thus, which which, under a broadest reasonable interpretation (MPEP § 2111), do not require a certain order.
interpreting the instruction by a virtual machine to perform the static initialization of the array.	<i>Lewis</i> discloses interpreting the instruction by a virtual machine, i.e., the "MCode interpreter," to perform the static initialization of the array.
,	"While the MCode interpreter is mostly platform-independent, about 20% of its code is platform-specific. For example, in order to fully support procedure interposition and other

U.S. Patent No. 6,061,520 - Claim 1	Lewis in view of Dyer and further in view of Proebsting
	ABI capabilities, the SPARC MCode interpreter does not directly interpret MCode ProcCall or Invoke instructions but instead implements them as SPARC ABI calls. Even MCode calls to other MCode procedures are implemented using SPARC instructions and execute the
	procedure's machine language entry code. This is necessary because the interpreter can never know whether a called procedure is actually implemented in MCode or in C. (For example, a programmer might have replaced the called procedure using interposition.) This means the intermeter must fully handle all the details required for ABI calls. If a called
	routine will return an aggregate value, the interpreter must generate a sequence of machine instructions at runtime (a <i>thunk</i>) to support the SPARC ABI's calling convention that the returned aggregate's length must be encoded into a SPARC UNIMP instruction just after the
	call. The interpreter also stores all program values in memory as SPARC values since this is required for ABI interoperation.
	Recently, a second MCode interpreter has been developed by Mick Jordan. This interpreter executes <i>system models</i> written in the Clarity language. These system models precisely describe how a software system is built: the exact versions of its component parts, all continue and build merameters and how the component parts are secondled. This system
	e.g. the inability to exactly reproduce the Construction of a software system. The system modeller's MCode interpreter is specialized to executing these models and to interacting with the Clarity program database. It does not need for example, to support ABI
	interoperation." Lewis at 127-28.

U.S. Patent No. 6,061,520 - Claim 1	Lewis in view of Dyer and further in view of Proebsting
	Inkable MC ode Generator MC ode manime Interpreter
	Figure 2: The mutime portion of the Clarity MCode system
	Lewis at 121.
U.S. Patent No. 6,061,520 – Claim 2	Lewis in view of Dyer and further in view of Proebsting
2. The method of claim 1 wherein the storing step includes step of: storing a constant pool entry into the constant	<i>Lewis</i> discloses the method of claim 1 wherein the storing step includes step of: storing a constant pool entry into the constant pool. For example, <i>Lewis</i> discloses the entry of a constant as a "CGValue."
.1000l	"The second C++ base class, CGValue, describes values during compilation. The code generator "executes" MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously "executed" subexpressions, and procedure or method calls." <i>Lewis</i> at 126.
	"This subsection presents an overview of the Clarity MCode compilation system; more details are given in Section 3. There are two major parts to the MCode compilation system: a program development-time part and a runtime part. The development-time portion consists of an MCode generator for the Clarity language and a Linkable MCode converter; see Figure 1. The MCode generator reads semantically decorated Clarity ASTs stored in the

U.S. Patent No. 6,061,520 - Claim 2	Lewis in view of Dyer and further in view of Proebsting
	Clarity programming environment's database and produces platform-independent MCode. The Linkable MCode converter then wraps a compact encoding of the MCode into a standard object file. Linkable MCode object files are then combined by standard linkers with other object files to produce executables and shared libraries."
	Editor Figure 1: The development-time portion of the Clarity MC ode object files
	<i>Lewis</i> at 120-21, Fig. 1.
	"Linkable MCode object files contain a machine-independent <i>pickle</i> of an MCode code unit. This is a compact, platform-independent encoding of the MCode information into a sequence of bytes. This pickle can later be internalized or unpickled to reconstruct the original MCode. The MCode for each procedure is pickled separately to support procedure-at-a-time processing. The current encoding is not especially compact although the Linkable MCode object files are still smaller than object files containing machine code. We intend to replace the current pickle format with a more compact one.
	Linkable MCode object files are platform-standard object files that are processed in the usual way by the platform's standard linker. This means they need to include platform-dependent definitions of global variables and procedures, and descriptions of referenced symbols. We currently encode ('mangle') symbol names in order to ensure that the resulting executables or shared libraries are type-safe with respect to the Clarity language. Eventually, this type-safety will be checked by a Clarity <i>prelinker</i> . Besides symbol definitions and references, our Solaris Linkable MCode object files also contain a few machine language instructions for each procedure's entry code. This entry code allows C code to call the MCode procedure. On the SPARC, this entry code consists primarily of a three instruction 'trampoline' that redirects the call to the appropriate target procedure

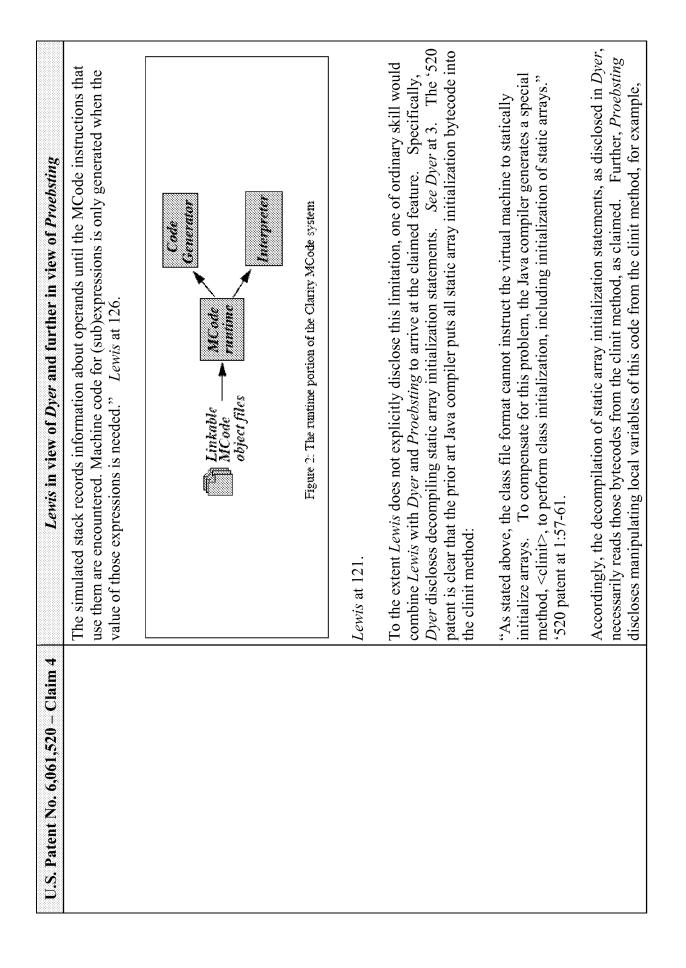
U.S. Patent No. 6,061,520 - Claim 2	Lewis in view of Dyer and further in view of Proebsting
	chosen by the interpret/compile strategy module in the MCode runtime. The SPARC entry code also has three words used when atomically updating the trampoline. Despite this platform-specific information, the contents of a Linkable MCode file are mostly platform-independent. The Linkable MCode converter itself is also mostly platform-independent. We currently execute the Linkable MCode converter during program development, before a program is distributed. It could also be executed when the program is installed on a particular platform." <i>Lewis</i> at 125-26.
U.S. Patent No. 6,061,520 - Claim 3	Lewis in view of Dyer and further in view of Proebsting
3. The method of claim 1 wherein	<i>Lewis</i> discloses the method of claim 1. <i>See</i> claim 1 chart above.
the play executing step includes the steps of: allocating a stack;	<i>Lewis</i> discloses the play executing step includes the steps of: allocating a stack. For example, <i>Lewis</i> discloses the play execution of MCode instructions to create CGValue entries that represent that state of the entries on the stimulated stack.
	"The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGV alue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." <i>Lewis</i> at 126.
	To the extent <i>Lewis</i> does not explicitly disclose this limitation, one of ordinary skill would combine <i>Lewis</i> with <i>Proebsting</i> to arrive at the claimed feature. Specifically, <i>Proebsting</i> discloses simulating the Java virtual machine stack:
	"Krakatoa uses a stack simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations." <i>Proebsting</i> § 1.

U.S. Patent No. 6,061,520 - Claim 3	Lewis in view of Dyer and further in view of Proebsting
	"Symbolic execution of the bytecode creates the corresponding Java source expressions Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." <i>Proebsting</i> § 2.
	One of ordinary skill in the art would understand that in order to simulate a stack, the <i>Proebsting</i> system necessarily must have "allocat[ed] a stack," as recited in this claim.
reading a byte code from the clinit method that manipulates the stack; and	<i>Lewis</i> discloses reading a byte code from the clinit method that manipulates the stack.
4	"The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants,
	variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." <i>Lewis</i> at 126.
	Linkable MC ode runnue
	Interpreter
	Figure 2: The number portion of the Clarity MCode system
	Lewis at 121.
performing the stack manipulation on	Lewis discloses performing the stack manipulation on the allocated stack. For example,

U.S. Patent No. 6,061,520 - Claim 3	Lewis in view of Dyer and further in view of Proebsting
the allocated stack.	<i>Lewis</i> discloses the generation of the machine code, which inherently performs stack manipulation on the allocated stack.
	"The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that
	use mem are encountered. Machine code for (subjexpressions is only generated when the value of those expressions is needed." Lewis at 126 .
	To the extent <i>Lewis</i> does not explicitly disclose this limitation, one of ordinary skill would combine <i>Lewis</i> with <i>Dyer</i> and <i>Proebsting</i> to arrive at the claimed feature. Specifically, this claim element would have been obvious at the time of the invention to one of ordinary skill in the art from the teachings of <i>Lewis</i> , either by itself or in combination with other relevant prior art, including, but not limited to the <i>Dyer</i> and <i>Proebsting</i> references.
	<i>Dyer</i> is a review of various Java decompilers, and <i>Proebsting</i> is a specific example of a Java decompiler. These decompilers convert Java class files into Java source code Running the <i>Proebsting</i> decompiler on the Java code disclosed in <i>Dyer</i> would lead to the "[s]ymbolic execution [that] simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." <i>Proebsting</i> at § 2. <i>Dyer</i> expressly identifies static array initializers, which would necessarily (because of the semantic rules of the Java language) be compiled as byte code in a clinit method of the Java class file:
	"Mocha transformed a static initializer into an elegant, but illegal, construction:
	<pre>public ConsoleWindow(String string, int i1) { dead = false;</pre>
	<pre>styles = { "Plain", "Bold", "Italic" };</pre>

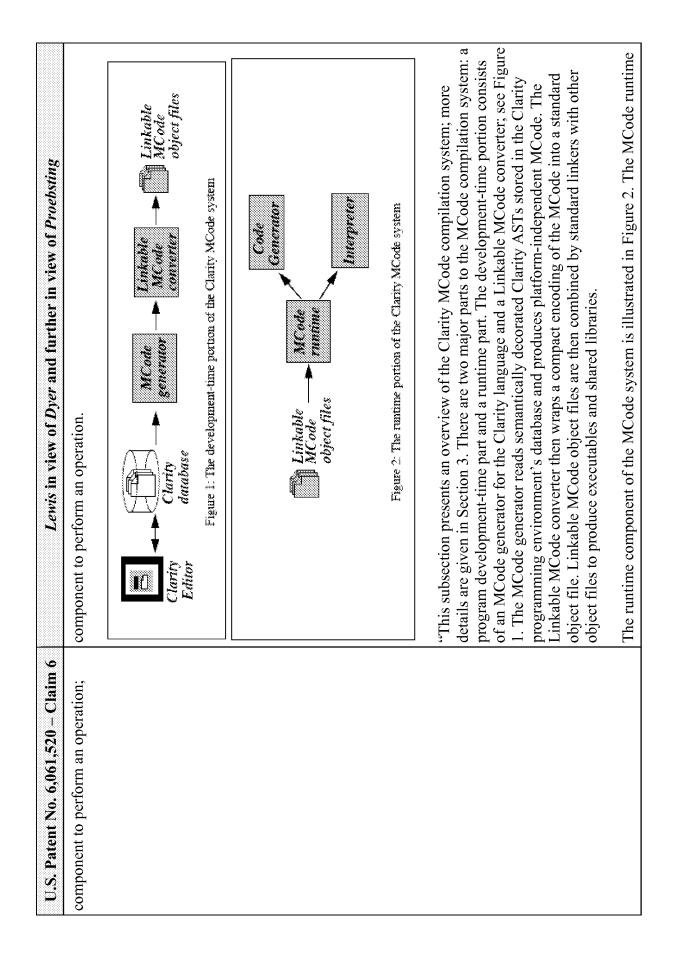
U.S. Patent No. 6,061,520 - Claim 3	Lewis in view of Dyer and further in view of Proebsting
	sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." $Dyer$ at 3.
	Thus, the combination of <i>Proebsting</i> and <i>Dyer</i> would result in "reading a byte code from the clinit method that manipulates the stack; and performing the stack manipulation on the allocated stack," as recited in Claim 3.
U.S. Patent No. 6,061,520 - Claim 4	Lewis in view of Dyer and further in view of Proebsting
4. The method of claim 1 wherein	<i>Lewis</i> discloses the method of claim 1. <i>See</i> claim 1 chart above.
the play executing step includes the steps of: allocating variables;	<i>Lewis</i> discloses the play executing step includes the steps of: allocating variables. For example, <i>Lewis</i> discloses the play execution of MCode instructions to create CGValue entries that represent that state of the entries on the stimulated stack.
	"The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGV alue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." <i>Lewis</i> at 126.

U.S. Patent No. 6,061,520 - Claim 4	Lewis in view of Dyer and further in view of Proebsting
	To the extent <i>Lewis</i> does not explicitly disclose this limitation, one of ordinary skill would combine <i>Lewis</i> with <i>Proebsting</i> to arrive at the claimed feature. Specifically, <i>Proebsting</i> discloses play executing (disclosed as "symbolic execution") that includes allocating variables:
	"Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed. For instance, iload_1, <i>which loads the value of the first local variable</i> —with type int—could be represented on the stack as 'i1'." <i>Proebsting</i> § 2 (emphasis added).
	"For the JVM dup operators, which duplicate stack elements, Krakatoa simply creates a temporary variable to hold the duplicated value Krakatoa again uses a temporary variable to hold the result of each branch of the conditional expression, and then assigns this temporary value to the conditional expression." <i>Proebsting</i> § 5.
	In light of <i>Proebsting</i> 's use of local variables, one of ordinary skill in the art would understand that they must necessarily be allocated, as recited. With respect to the "dup" operator (quoted above), it is particularly noteworthy that the '520 patent discloses that this operation is inherently part of the code that is generated in the static array initialization bytecode. <i>See</i> '520 patent at 2:31; <i>see also</i> '520 patent at 5:25-43 (Code Table #4 disclosing that "dup" is one of the bytecodes that is recognized by the preloader). Since <i>Proebsting</i> was decompiling Java bytecode, it necessarily would have allocated a local variable when encountering the "dup" bytecode that is part of the static array initialization.
reading a byte code from the clinit method that manipulates local variables of the clinit method; and	<i>Lewis</i> discloses reading a byte code from the clinit method that manipulates local variables of the clinit method. For instance, <i>Lewis</i> discloses reading MCode instructions, which may include initialization instructions.
	"The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls.



U.S. Patent No. 6,061,520 – Claim 4	Lewis in view of Dyer and further in view of Proebsting
	when simulating the "dup" bytecode:
	"For the JVM dup operators, which duplicate stack elements, Krakatoa simply creates a temporary variable to hold the duplicated value Krakatoa again uses a temporary variable to hold the result of each branch of the conditional expression, and then assigns this temporary value to the conditional expression." <i>Proebsting</i> § 5.
performing manipulation of the local variables on the allocated variables.	<i>Lewis</i> discloses performing manipulation of the local variables on the allocated variables. For example, <i>Lewis</i> discloses the manipulation of code to create CGValues.
	"The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." <i>Lewis</i> at 126.
	To the extent <i>Lewis</i> does not explicitly disclose this limitation, one of ordinary skill would combine <i>Lewis</i> with <i>Dyer</i> and <i>Proebsting</i> to arrive at the claimed feature. Specifically, one of ordinary skill in the art would understand that in order to manipulate the local variables, as disclosed in <i>Proebsting</i> , such manipulation would necessarily occur on the allocated variables.
U.S. Patent No. 6,061,520 – Claim 6	Lewis in view of Dyer and further in view of Proebsting
6. A method in a data processing system, comprising the steps of:	<i>Lewis</i> discloses a method in a data processing system, e.g., the C++ language and certain dialects thereof, capable of performing data processing.
	"The <i>Clarity</i> C^{++} programming language is a dialect of C^{++} being developed in Sun Microsystems Laboratories. Clarity shares many features with C^{++} but is less complex and has a more consistent syntax and simpler semantics without loss in expressiveness

U.S. Patent No. 6,061,520 - Claim 6	Lewis in view of Dyer and further in view of Proebsting
	Clarity is intended to be a wide-spectrum language suitable for both systems and application programming, particularly of distributed software.
	To support the compilation of Clarity, we have developed a high-level, machine-independent intermediate representation that we call <i>MCode</i> (for "middle code"). We use MCode to compile Clarity programs at execution time (i.e., on-the-fly) into SPARC code for the Solaris operating system. This code generator is designed to be largely machine independent: besides the SPARC code generator, an Intel x86 version is being developed." <i>Lewis</i> at 119 (footnote omitted).
	"MCode has its basis in unpublished work done by L. Peter Deutsch at Sun Microsystems Laboratories in 1992-93. This work consisted of an implementation in Smalltalk of the core of a portable, on-the-fly compiler for a subset of the C language; we will refer to this system as 'CCore." <i>Lewis</i> at 120.
	<i>Dyer</i> discloses a review of three Java Decompilers. These are programs that convert Java class files into Java source code, effectively reverse engineering compiled code to figure out how the underlying code works. <i>Dyer</i> discloses, in most relevant part, an example of the code that would be used to implement a static initialization of an array. <i>Dyer</i> not only discloses the concept of the static initialization of an array by way of an instruction provided to the Java virtual machine; <i>Dyer</i> also provides examples of actual code to achieve this result.
	The decompiler of $Dyer$ inherently performs a symbolic execution of the compiled bytecode. The functionality of such a decompiler, including the symbolic execution that decompilers use to identify the target of compiled bytecode for the recovery of the underlying Java source code, is disclosed in detail in <i>Proebsting</i> . The combination of the <i>Proebsting</i> disclosure, which explains the symbolic execution technique of a decompiler, with the actual results of a decompiler, as shown by <i>Dyer</i> , render obvious the broad symbolic execution of claim 6.
receiving code to be run on a processing	Lewis discloses receiving code, i.e., Linkable MCode object files, to be run on a processing



U.S. Patent No. 6,061,520 - Claim 6	Lewis in view of Dyer and further in view of Proebsting
	in an MCode-containing executable internalizes the MCode for each procedure as needed, when the procedure is first called. It also implements a interpret/code generate policy separately for each MCode procedure. This policy chooses for each procedure whether to interpret it, generate code, or interpret then later generate code, or generate better code. The code generator currently produces SPARC code of approximately the quality of the SunPRO C compiler at the default -O2 optimization level. A port of the code generator to the x86 is underway. The MCode interpreter interoperates with all SPARC ABI code. Like the
	computer, it is recuttant and supports multitureaded programs. It also does extensive checking during program execution, which makes it especially useful for uncovering errors in Clarity programs that are otherwise difficult to detect. The interpreter will also be used by the Clarity debugger that we are developing to evaluate Clarity statements and expressions." <i>Lewis</i> at 120-21.
	To the extent <i>Lewis</i> does not explicitly disclose this limitation, one of ordinary skill would combine <i>Lewis</i> with <i>Proebsting</i> to arrive at the claimed feature. Specifically, <i>Proebsting</i> discloses receiving Java byte code that was compiled to run on a processing component:
	"This paper presents our technique for automatically decompiling Java bytecode into Java source." <i>Proebsting</i> at Abstract.
	" <i>Decompilation</i> transforms a low-level language into a high-level language. The Java Virtual machine (JVM) specifies a low-level bytecode language for a stack-based machine. This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a <i>class file</i> that contains type information and bytecode. The JVM requires a significant amount of type information from the class files for object linking Decompilation systems can exploit this type of information and well-behaved property to recover Java source code from the class file.
	We present a technique for transforming low-level Java bytecode into legal Java source code." <i>Proebsting</i> § 1.

U.S. Patent No. 6,061,520 - Claim 6	Lewis in view of Dyer and further in view of Proebsting
	"We have implemented a prototype Java decompiler, Krakatoa, in Java." Proebsting § 5.
play executing the code without running the code on the processing component to identify the operation if the code were run by the processing component; and	<i>Lewis</i> discloses play executing the code without running the code on the processing component to identify the operation if the code were run by the processing component. For example, <i>Lewis</i> discloses simulating execution of the MCode instructions in the MCode object files, and that the MCode instruction set includes instructions for statically initializing arrays.
	"The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously "executed" subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." <i>Lewis</i> at 126.
	The simulation of byte codes against a memory to identify underlying code, such as the static initialization of an array, was commonly known to those of ordinary skill in the art at the time of the invention. For example, the decompiler technology disclosed by <i>Proebsting</i> discusses "decompiling Java bytecode into Java source." <i>Proebsting</i> at Abstract. Specifically: "[s]ymbolic execution of the bytecode creates the corresponding Java source expressions." <i>Id.</i> at § 2. This decompilation method works because "[s]ymbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being compounded." <i>Id.</i>
	To the extent <i>Lewis</i> does not explicitly disclose this limitation, one of ordinary skill would combine <i>Lewis</i> with <i>Dyer</i> and <i>Proebsting</i> to arrive at the claimed feature. Specifically, <i>Dyer</i> and <i>Proebsting</i> together show that the play execution of byte codes against a memory to identify underlying code, such as the static initialization of an array, was commonly known to those of ordinary skill in the art at the time of the invention. For example, the decompiler technology disclosed by <i>Proebsting</i> discloses a technique for decompilation of Java byte code that play executes the Java virtual machine's stack:

U.S. Patent No. 6,061,520 - Claim 6	Lewis in view of Dyer and further in view of Proebsting
	"This paper presents our technique for automatically decompiling Java bytecode into Java source." <i>Proebsting</i> at Abstract.
	" <i>Decompilation</i> transforms a low-level language into a high-level language. The Java Virtual machine (JVM) specifies a low-level bytecode language for a stack-based machine. This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a <i>class file</i> that contains type information and bytecode. The JVM requires a significant amount of type information from the class files for object linking Decompilation systems can exploit this type of information and well-behaved property to recover Java source code from the class file.
	We present a technique for transforming low-level Java bytecode into legal Java source code." <i>Proebsting</i> § 1.
	"We have implemented a prototype Java decompiler, Krakatoa, in Java." Proebsting § 5.
	"Krakatoa uses a stack simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations." <i>Proebsting</i> § 1.
	"Symbolic execution of the bytecode creates the corresponding Java source expressions Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." <i>Proebsting</i> § 2.
	Thus it would have been obvious to one of ordinary skill in the art to combine the methods of $Lewis$ with the play execution (symbolic execution) of bytecode against a memory, as disclosed by <i>Proebsting</i> . And, even more specifically, the artisan would look to known decompilers, such as those disclosed in <i>Dyer</i> . <i>Dyer</i> reviews three popular decompilers that were available before the '520 patent was filed: DejaVu, Mocha, and WingDis. <i>See Dyer</i> at 1. In that review, <i>Dyer</i> analyzes the decompilation of code, including the

U.S. Patent No. 6,061,520 - Claim 6	Lewis in view of Dyer and further in view of Proebsting
	decompilation of static initializers:
	"Mocha transformed a static initializer into an elegant, but illegal, construction:
	<pre>public ConsoleWindow(String, int il) {</pre>
	<pre>dead = false; styles = { "Plain", "Bold", "Italic" }; sizes = { "8", "9", "10", "12", "14", "16", "18", "24" }; </pre>
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." <i>Dyer</i> at 3.
	The fact that some of <i>Dyer's</i> decompiled code may be considered "illegal" should not detract from the fact that the decompilation of static array initialization byte code into a single expression or instruction was a well-known technique at the time the '520 patent was filed a technique used to identify the operation of the byte code in order to generate equivalent high-level language expressions. In fact, the '520 patent discloses creating a constant pool entry that is not a standard Java virtual machine construct. Thus, one of ordinary skill in the art would have considered this constant pool construct (i.e., CONSTANT_Array (<i>see</i> '520 patent at 8:54-9:13)) as an "illegal" Java constant pool entry. The reason it wouldn't have mattered is that the '520 patent states that "the virtual machine 222 is modified to recognize the static initialization directives of the preloader." '520 patent at 4:46-48. Because the exact form of modification is not disclosed, it must be within the ordinary artisan's skill set to perform the correct modification to allow the virtual

U.S. Patent No. 6,061,520 - Claim 6	Lewis in view of Dyer and further in view of Proebsting
	machine to recognize an "illegal" instruction such as the new CONSTANT_Array type.
	Further, during Examination claim terms are to be given their broadest reasonable interpretation (<i>see</i> MPEP § 2111), and there are no qualifiers in the claim language as to the
	generated instruction. Again, because the '520 patent assumes that one of ordinary skill in the art would modify the virtual machine to recognize the generated instruction, the exact
	format (whether syntactically correct or not) is not an issue. And in any case, <i>Dyer</i> discloses that in contrast to the Mocha decompiler's "illegal" code the WingDis decompiler produced "syntactically correct code." <i>Dyer</i> at 3.
creating an instruction for the processing component to perform the	<i>Lewis</i> discloses creating an instruction, <i>e.g.</i> , "MCode," for the processing component to perform the operation.
	"The object-oriented architecture of the code generator has significantly simplified our implementation. The MCode machine code generator is designed to be retargetable to a new machine architecture (especially a RISC machine) with relatively little effort. It defines two key C++ base classes that must be subclassed to nort the code generator. The first class
	CGMachine, represents a target machine for code generation and a code stream for that machine. The basic machine model is a generic, nonwindowed RISC processor. CGMachine subclasses may define variations such as windowed RISC and CISC. These subclasses
	implement virtual methods that describe the target machine's registers, data types, and instruction properties. CGMachine methods then use those descriptions to generate machine code from MCode
	The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants,
	The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." <i>Lewis</i> at 126.

U.S. Patent No. 6,061,520 - Claim 6	Lewis in view of Dyer and further in view of Proebsting
	"In order to do high-quality code generation, we need to rebuild the expression trees from the stack machine. The code generator defers generation until the final target for an expression is known. Much of the process is similar to that of TNBIND algorithm documented in [Wulf 75]. This algorithm gives excellent results and executes extremely efficiently. A final optimization does instruction reordering to minimize RISC processor pipeline execution conflicts. By organizing the code generator into a series of cascading object streams, we are able to consume MCode and generate native machine code in one pass. Our object-oriented architecture provides an efficient way to trade increased memory for speed." <i>Lewis</i> at 127.
	"We have described an intermediate representation MCode that is compact, easy to generate, and supports the on-the-fly generation of good quality machine code. Linkable MCode is an encoding of MCode in platform-standard object files that enables full interoperation with C and existing libraries, as well as the full use of all capabilities of standard linkers and other programming tools." <i>Lewis</i> at 128.
	To the extent <i>Lewis</i> does not explicitly disclose this limitation, one of ordinary skill would combine <i>Lewis</i> with <i>Proebsting</i> and <i>Dyer</i> to arrive at the claimed feature. Specifically, <i>Proebsting</i> discloses that the Krakatoa decompiler writes the source code (which would necessarily include creating any static array initialization instructions) to an output file:
	"We have implemented a prototype Java decompiler, Krakatoa, in Java. We have run Krakatoa on a number of class file, including some to which we had no source code access. We examined the output of Krakatoa by hand, and Krakatoa appears to recover high-level constructs very well." <i>Proebsting</i> § 5.
	"Krakatoa is very efficient at reproducing readable Java source from Java bytecode." Proebsting § 6.
	<i>Dyer</i> discloses that the output of a decompiler (such as <i>Proebsting</i> 's Krakatoa decompiler) may be an instruction that "request[s] the static initialization of the array," as recited in the

U.S. Patent No. 6,061,520 - Claim 6	Lewis in view of Dyer and further in view of Proebsting
	claim:
	"Mocha transformed a static initializer into an elegant, but illegal, construction:
	public ConsoleWindow(String string, int il)
	dead = false; styles = { "Plain", "Bold", "Italic" }; sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." <i>Dyer</i> at 3.
	The fact that some of $Dyer$'s decompiled code may be considered "illegal" should not detract from the fact that the decompilation of static array initialization byte code into a single expression or instruction was a well-known technique at the time the '520 patent was filed a technique used to identify the operation of the byte code in order to generate equivalent high-level language expressions. In fact, the '520 patent discloses creating a
	constant pool entry that is not a standard Java virtual machine construct. Thus, one of ordinary skill in the art would have considered this constant pool construct (i.e., CONSTANT_Array (<i>see</i> '520 patent at 8:54-9:13)) as an "illegal" Java constant pool entry.
	The reason it wouldn't have mattered is that the '520 patent states that "the virtual machine 222 is modified to recognize the static initialization directives of the preloader." '520 patent at 4:46-48. Because the exact form of modification is not disclosed, it must be within the ordinary artisan's skill set to perform the correct modification to allow the virtual

U.S. Patent No. 6,061,520 - Claim 6	Lewis in view of Dyer and further in view of Proebsting
	machine to recognize an "illegal" instruction such as the new CONSTANT_Array type.
	Further, during Examination claim terms are to be given their broadest reasonable interpretation (<i>see</i> MPEP § 2111), and there are no qualifiers in the claim language as to the generated instruction Δ as in because the '500 metant assumes that one of ordinary shift in
	format (whether syntactically correct or not) is not an issue. And in any case, <i>Dyer</i> discloses that in contrast to the Mocha decompiler's "illegal" code the WingDis decompiler produced "syntactically correct code "Dyer at 3
	Additionally, the recitation of this method step in Claim 1 of the '520 patent is not limited with respect to the order in which it is performed relative to the other steps of Claim 1. Thus, the combined systems of <i>Lewis</i> , <i>Proebsting</i> , and <i>Dyer</i> may perform this step (i.e.,
	"creating an instruction") after it performs the step of "receiving code" and before the step of "reading the byte code." Or it may perform this step after "reading the byte code." Either
	scenario would read on the claim limitations, which, under a broadest reasonable interpretation (MPEP § 2111), do not require a certain order.
U.S. Patent No. 6,061,520 – Claim 7	Lewis in view of Dyer and further in view of Proebsting
7. The method of claim 6 wherein the operation initializes a data structure.	<i>Lewis</i> discloses the method of claim 6 wherein the operation initializes a data structure. The MCode includes at least instructions and data structures, as shown below.
and	
	Although MCode includes instructions and data structures needed to implement some Clarity language-specific constructs such as its exceptions and method calls, the core of MCode is suitable for representing code for C and many other languages." <i>Lewis</i> at 119.

U.S. Patent No. 6,061,520 - Claim 7	Lewis in view of Dyer and further in view of Proebsting	oebsting
	AllocArray(array type index) Begin	BeginSwitch(end swi
	ArrayIndex(array type index)	EndSwitch(end switc
		BeginExprCase(int)
	883µ	BeginDefaultCase()
	InvokeOuter(method type index) EndC	EndCase(end switch
	InvokeDe/sgated(method type index)	
	Widen(base interface number)	DoBreak(end tag)
	Narrow(tanget type index)	DoContinue(end tag)
	Figure 4: The MCode instruction set	inction set
	Lewis at 123 Fig. 4 (excerpt).	
	"Figures 5 and 6 give an example of generated MCode. The Clarity method <i>startup</i> in Figure 5 produces the MCode instructions shown in part in Figure 6."	y method <i>startup</i> in

				ieft_sibiling: Worker; // left delegates; manages "left_workers" workers right_sibiling: Worker; // right delegates; manages "right_workers" workers	Let.	our_work: int = 0; seft workers: int = 0;	to_work: method (work_to_do: in int) { }* exided */ };	// the following declarations are private to the WorkerImpi implementation	$(M_1, \dots, M_{n-1}, \dots, \dots,$	ThreadedSimulation: module		U.S. Patent No. 6,061,520 – Claim 7 Lewis in view of Dyer and further in view of <i>Prochsting</i> ThreadedSmutakon: module work, partex Threads Mulex, work, partex threads Mulex, work, partex in the intervention of the interventin of the intervention of the intervention
					ien_sisiing: worker, right_sisiing: Worker,	right_workers: int = 2; ieft_sibing: Worker; กฎก_รibing: Worker;	ວມr_work: irst = 0; ieft_workers: int = 2; right_workers: int = 2; ieft_sibing: Worker; right_sibing: Worker;	ag_work. meanue (work_m_goo. m our_workers. int = 0; nght_workers. int = 0; nght_sibing: Worker; nght_sibing: Worker;	inpleme inpleme { f f f f f f f f f f f f f f f f f f	off_mutex: off_work: if for kimpt: type in pleme: in pleme: in the foll of work: if sith	ThreadedSimulation: module work, mutex. Threade: MMex, work, mutex. Threade: MMex, work, mutex. Threade: MMex, work, mutex. Threade: Mexels: Threads: Mexels: Mexe	our_work: nz = u; teft_workers: int = 2; right_workers: int = 2; teft_sitting: Worker; right_sitting: Worker;
<pre>// the following declarations are pr do_work: method (work_to_do: in our_work: int = 0; ieft_workers: int = 2; right_workers: int = 2; right_workers: int = 2; right_sibling: Worker; nght_sibling: Worker;</pre>	<pre>// the following declarations are pr do_work: method (work_to_do: in our_workers: int = 0; ieft_workers: int = 2; right_workers: int = 2; ieft_sibling: Worker; nght_sibling: Worker;</pre>	 If the following declarations are production do work: method (work_to_do: in our_work: int = 0; ieft_workers: int = 0; ieft_workers: int = 0; ieft_sibing: Worker; right_sibing: Worker; right_sibing: Worker; 	 If the following declarations are private to the WorkerImpl implementation do_work: method (work_to_do: in int) { /* elided */ }; Our_work: int = 0; Self_workers: int = 0; 	<pre>// the following declarations are private to the WorkerImpi implementation do_work: method (work_to_do: in int) { /* elided */ }; cur_work: int = 0; ieft_workers: int = 0; right_workers: int = 0;</pre>	// the following declarations are private to the WorkerImpf implementation do_work: method (work_to_do: in int) { /* elided */ }; our_work: int = 0; set environ: int = 0;	// the following declarations are private to the Workertinpi implementation do_work: method (work_to_do: in int) { // elided */ };	// the following declarations are private to the Workerting/ implementation		<pre>vommap: type = implementation on workers: in int) implement startup: method (our_workers: in int) {</pre>	<pre>work_mutex: Threads: Mutex; work_per_worker: Int:</pre>	ThreadedSimulation: module work_matter: Threader: Muter; work_per_worker: int: // protected by work_muter; work_per_worker: int: // protected by work_muter; work_per_worker: int: // protected by work_muter; work_per_works: Thread 3; // an unusual interface: no methods beyond Thread.:tstrtup and the other Thread methods // an unusual interface: no methods beyond Thread.:tstrtup and the other Thread methods // an unusual interface: no methods beyond Thread.:tstrtup and the other Thread methods // // a executed when the thread is tstrted; defegates most of the within statement our work = work_per_workers: in i cour, workers: in it (a event work = 1, a facture work_muter tor duration of the within statement // (a event = 0) (our_workers = 1, a fact_workers; if (adlegated_lo_workers > 0) (our_workers = 1, a fact_workers; if (adlegated_lo_workers > 0) (adregated_lo_workers > 0) (adregated_lo_workers > 0) (adregated_lo_workers); // (adlegated_lo_workers > 0) (adregated_lo_workers); // (adregated_lo_workers > 0) (adregated_lo_workers > 0); // (adregated_lo_workers > 0) (adregated_lo_workers); // (adregated_lo_workers); // (adregated_lo_workers); // (adregated_lo_workers); // (adregated_lo_workers); // (adregated_lo_workers); // (adregated_lo_workers); // (adregated_lo_workers); // (adregated_lo_w	
<pre>% % the following declarations are pr do_work: method (work_to_do: in cur_work: int = 0; % fight_workers: int = 0; % order; %</pre>	<pre>% % % % % % % % % % % % % % % % % % %</pre>	ג לו להפינוסאות לא לא לא לא היא לא היא היא לא לע_work: method (שטרג_נמ_לס: וח סטר_שטרגר: ווז = 0; נפות_שטרגרר: ווז = 1; נפות_פוטוות: Worker; נפות_פוטוות: Worker;	 If the following declarations are private to the WorkerImpl implementation do_work: int = 0; our_work: int = 0; ieft_workers: int = 0; ieft_stitht_workers: int = 0; ieft_stitht; Worker; int defeates; manages "eft_workers" workers 	If the following declarations are private to the WorkerImpl implementation do_work: method (work_to_do: in int) { /* elided */ }; our_work: int = 0; left_workers: int = 0; right_workers: int = 0;	If the following declarations are private to the WorkerImpl implementation do_work: method (work_to_do: in int) { /* exided */ }; out_work: int = 0; set workerer int = 0;	// the following declarations are private to the WorkerImpi implementation do_work: method (work_to_do: in int) { /* elided */ };); the following declarations are private to the WorkerImpt implementation		<pre>workingp:type = implement startup: method (our_workers: in int) inplement startup: method (our_workers: in int) i = executed when the thread is started; delegates most of its work to forked sitting workers within work_mutex { // acquire work_mutex for duration of the within statement within work_mutex { // acquire work_mutex for duration of the within statement within work = work_per_work = 1; extra_work = 1;) if (elegated_to_workers int = (our_workers - 1); if (delegated_to_workers > 0) (our_workers - 1); if (delegated_to_workers > 0) int_solvers? if (left_workers = delegated_to_workers.); if (left_workers = delegated_to_workers.); if (left_workers = delegated_to_workers.); if (left_workers = delegated_to_workers.); if (left_workers > 0) int_solvers); if (left_workers); if (left_workers) = left_workers); if (left_workers)</pre>	<pre>// work_mutex.Threads.Mutex, // perceded by work_mutex work_per_worker: int: // protected by work_mutex work_per_worker: int: // protected by work_mutex work_more.th; // protected by work_mutex // an unusual interface: no methods heycord Thread::startup and the other Thread methods // an unusual interface: no methods heycord Thread::startup and the other Thread methods // an unusual interface: no methods heycord Thread::startup and the other Thread methods // an unusual interface: no methods heycord Thread::startup and the other Thread methods // an unusual interface: no methods heycord Thread::startup and the other Thread // an unusual interface: no methods heycord Thread::startup and the other Thread // an unusual interface: no method sur workers: in int // an unusual interface: no method sur work = 1, extra_work = not, pert_worker // and, pert_work = not, pert_work = 1, extra_work = 1; ext</pre>	ThreadedSimulation: module <pre></pre>	
do_work(our_work); }; // the following declarations are pr do_work: int = 0; eur_work: int = 0; ight_workers: int = 2; right_workers: int = 2; right_workers: int = 2; right_workers: int = 2; right_sibiling: Worker; 	<pre> g do_work(our_work); } f the following declarations are pr do_work: inst = 0; eff_workters: inst = 0; ieff_workters: inst = 0; ieff_workters: inst = 0; ieff_eibling: Workter; ind = 0; ieff_eibling: Workter; } </pre>	} do_work(our_work); // the following declarations are pr do_work: int = 0; our_workers: int = 2; feft_workers: int = 2; feft_sibing: Worker; right_sibing: Worker;	از the following declarations are private to the WorkerImpl implementation If the following declarations are private to the WorkerImpl implementation do_work: int = 0; eff_workers: int = 0; infl_workers: int =	غرار الله الله الله الله الله الله الله ا	غرف work(our_work); از the toilowing declarations are private to the WorkerImpi implementation do_work: method (work_to_do: in int) { // elided */ }; our_work: int = 0; add_workerer int = 0;	غ الا عنه المحمد ال الا عنه المحمد المحم عنه المحمد ال	غ التالية المراجعة الم المراجعة المراجعة الم		<pre>working: type = imperation of workers: in int) implement startup: method (our_workers: in int)</pre>	<pre>work_mutax: Threads.Mutex; work_mutax: Threads.Mutex; work_per_worker: int; // protected by work_mutex work_per_worker: int; // protected by work_mutex Workmpi: type = imterface inherits Threads::Thread 9; // an unusual interface: no methods beyond Thread::tartup and the other Thread methods Workmpi: type = implementation of Worker implement startup: method (our_workers: in int); // // acquire work_mutex (// acquire work_mutex for duration of the within statement our_work = work > 0 (our_workers: int = (our_workers - 1;); // (delegated_fo_workers: int = (our_workers - 1;); // (delegated_fo_workers = delegated_fo_workers: int if(delegated_fo_workers = delegated_fo_workers); // (fit #workers = delegated_fo_workers); // (fit #workers); // (fit #workers);</pre>	ThreadedSimulation: module % ork_muter: Threads: Muter; work_per_worker: INt: // protected by work_mutek work_per_worker: INt: // protected by work_mutek extra_work: int: // protected by work_mutek extra_work: int: // protected by work_mutek f an unusual interface: no methods beyond Thread::tartup and the other Thread nethods Workinp: type = implement startup: method (our_workers: in int) { // excuted when the thread::tartup and the other Thread nethods mithin work = work // acquire work_mutek for duration of the writin statement our_work = work // acquire work_mutek for duration of the writin statement if (celegated_to_workers: in int) } } f (inf) workers = delegated_to_workers > 0; if (celegated_to_workers = delegated_to_workers); if (fell_workers = delegated_to_workers); if (fell_workers = delegated_to_workers); if (fell_workers = delegated_to_workers); if (fell_workers = delegated_to_workers); }	nght sibling = new Workerknpkinght_workers)
right_sibling = new / do_work(our_work); // the following declarations are pr do_work: int = 0; do_work: int = 0; left_workers: int = 2; right_workers: int = 2; right_workers: int = 2; right_workers: int = 2; right_workers: int = 2; right_sibling: Worker;	right_sibling = new / do_work(our_work); // the following declarations are pr do_work: method (work_to_do: in do_work: int = 0; left_workers: int = 2; right_workers: int = 2; right_workers: int = 2; right_sibling: Worker; night_sibling: Worker;	nght_sibling = new / do_work(our_work); // the tollowing declarations are pr do_work: method (work_to_do: in do_work: nt = 0; left_workers: int = 0; left_sibling: Worker; right_sibling: Worker;	inglet_sibling = new Workerinp(ingint_workers); it do_work(our_work); it eff_work; int = 0; it it it it it it it it it do it do it it it	ight_sibling = new Workerinp(nght_workers); do_work(our_work); // the following declarations are private to the WorkerImpl implementation do_work: method (work_to_do: in int) { /* elided */ }; our_work: int = 0; left_workers: int = 0; right_workers: int = 0;	ight sibling = new Workerinnp(right workers); do_work(our_work); If the following declarations are private to the WorkerImpi implementation do_work: int = 0; our_work: int = 0; add_workere_int = 0;	ight_sibling = new Workerinp(nght_workers); do_work(our_work); If the tollowing declarations are private to the WorkerImpi implementation do_work: method (work_ta_do: in int) { /* exided */ };	<pre>ight_sibling = new Workerfnip(right_workers);</pre>	ر م	<pre>working: type = imperientation of workers: in int) { implement startup: method (our_workers: in int) if executed when the thread is started; delegates most of its work to forked stbling workers within work_mutex { if acquire work_mutex for duration of the within statement our_work = work_per_worker; if (extra_work > 0) (our_work += 1; extra_work -= 1;) delegated to workers: int = (our_workers - 1); if (delegated to workers > 0) { if (delegated to workers = delevated to workers - 0); } }</pre>	<pre>work_mutex: Threads: Mutex; work_per_worker: int;</pre>	ThreadedSimulation: module work_per_worker: int: // protected by work_mutex work_per_worker: int: // protected by work_mutex work_per_work: int: // protected by work_mutex work.per_work: Threads: Threads: Threads: 1 // an unusual interface: no methods beyond Threads::thrup and the other Thread methods Workmpi: type = implementation of Worker: // an unusual interface: no method (our_workers: in int) // a secuted when the thread is started; delegates most of its work to forked staing workers within work_mutex { // acquire work_mutex for duration of the within statement our_work = work_per_worker: in int) // elegated [o_workers: int = (our_workers: int = (our_workers - 1); // delegated [o_workers: int = (our_workers - 1); // delegated [o_workers = dolework] // uncers).	fight workers - between setting and and a setting and and a setting and and a setting an
<pre>ight_workers = dele if (left_workers = dele if (left_workers = dele if (left_workers = dele if the following declarations are pr do_work: int = 0; ieft_workers: int = 0; ieft_workers: int = 0; ieft_sibing: Worker; ieft_sibing: Worker;</pre>	<pre>ight_workers = dele if (left_workers = dele if (left_workers = dele if (left_workers = dele if the following declarations are pr do_work: int = 0; if the following declarations are pr do_work: int = 0; ieft_workers: int = 0; ieft_sibling: Worker; ieft_sibling: Worker;</pre>	ent_workers = dele if (left_workers = dele if (left_workers = dele if (left_workers = dele if the following declarations are pr do_work: int = 0; do_work: int = 0; left_workers: int = 0; left_staing: Worker; right_workers: int = 0; left_staing: Worker; right_staing: Worker;	<pre>applie and applied and applied and applied and applied ap</pre>	<pre>continue contains = delegated_to_workers; fight_workers = delegated_to_workers; if (left_workers = 0) isft_sibling = new Workerfinp(inght_workers); is do_work(our_work); } do_work(our_work); } if the following declarations are private to the Workerfinp(implementation do_work: nethod (work_to_do: in int) { /* elided */ }; our_work: int = 0; ieft_workers: int = 0; ieft_workers: int = 0; isft_workers: int = 0;</pre>	<pre>applit_workers = delegated_io_workers; if (left_workers = delegated_io_workers;) if (left_workers > 0) keft_sibling = new WorkerImpi(left_workers); if a_work(our_work); } do_work(our_work); } if the following declarations are private to the WorkerImpi implementation do_work: method (work_to_do: in int) { /* elided */ }; our_work: int = 0; ieft_workers int = 0;</pre>	<pre>infit_workers = delegated_to_workers; if (left_workers > 0) left_sibling = new Workertimpi(left_workers); ight_sibling = new Workertimpi(nght_workers); do_work(our_work); } // the toilowing declarations are private to the WorkerImpi implementation do_work: method (work_to_do: in int) { /* elided */ };</pre>	<pre>infit workers = underset to workers; if (left workers > 0) left_sibling = new Workers have (left_workers); if (left_workers > 0) left_sibling = new Workers); do_work(our_work); } if the following declarations are private to the WorkerImpi implementation</pre>	ر م	<pre>working: type = impenentation of workers: in int) { implement startup: method (our_workers: in int) /</pre>	<pre>work_mutex: Threads: Mutex; work_per_worker: int;</pre>	ThreadedSimulation: module work_per_worker: int;	ii (delegated to workers > 2) {
if (delegated to workers ieft, workers = deleg right_workers = deleg right_sibling = new \ if (left_workers > 0) right_sibling = new \ do_work(our_work); if the following declarations are pr do_work(our_work); if the following declarations are pr do_work; if the following declarations are pr do_work; do_work; if the following declarations are pr do_work; do_work; do_work; if the following declarations are pr do_work; do_wo	if (delegated to workers ieft, workers = deleg if (left, workers = deleg if (left, workers > 0)) inght_sibling = new \ if (left_workers > 0) inght_sibling = new \ do_work(our_work); if (left_workers int = 0; if the workers: int = 0; if the wo	<pre>if (delegated fo_workers if (delegated fo_workers = deleg if (left_workers = dele if (left_workers > 0)) ight_sibling = new \</pre>	<pre>if (delegated io_workers > 0) { if (delegated io_workers); if (delegated io_workers); if (left_workers = delegated io_workers); if (left_workers); if (l</pre>	<pre>if (delegated /o_workers > 0) {</pre>	<pre>if (delegated io_workers > 0) { if (delegated io_workers > 0) { if (eff_workers = delegated_io_workers): ight_workers = delegated_io_workers): ight_sibling = new Workers > 0) ieft_sibling = new Workers); ight_sibling = new Workers > 0) ieft_sibling = new Workers); do_work(our_work); if (left_workers > 0) ieft_sibling = new Workerting(ifft_workers); if the following declarations are private to the Workerting implementation do_work: method (work_is, int = 0; our_work: int = 0; if the following declarations are private to the Workerting implementation do_work: int = 0; if the following declarations are private to the Workerting implementation do_work: int = 0; if the following declarations are private to the Workerting implementation do_work: int = 0; if the following declarations are private to the Workerting implementation do_work: int = 0; do_work: int = 0; do_work int = 0; do_work int = 0; do_work int = 0; do_workere int = 0;</pre>	<pre>if (delegated io_workers > 0) { if (delegated io_workers > 0) { if the workers = delegated_io_workers/_ if (ieft_workers = delegated_io_workers/_ if (ieft_workers = delegated_io_workers/_ if (ieft_workers > 0) ieft_sibling = new Workerfmpi(ieft_workers); ight_sibling = new Workerfmpi(ight_workers); ight_sibling = new Workerfmpi(ight_workers); if (the following declarations are private to the WorkerImpi implementation do_work: method (work_io_do: in int) { /* elded */ }; } </pre>	<pre>if (delegated io_workers > 0) { if (delegated io_workers = delegated io_workers): if (left_workers = delegated io_workers.): if (left_workers = delegated io_workers.): if (left_workers > 0) left_sibling = new Workerimpi(left_workers); if the following declarations are private to the Workerimpi implementation</pre>	ा दुर्व पुर्व	<pre>workings: type = implementation of workers: in int) { inplement startup: method (our_workers: in int) // executed when the thread is started; delegates most of its work to forked sibling workers within work_mutex { // acquire work_mutex for duration of the within statement our_work = work_per_work = 1; extra_work -= 1; }</pre>	<pre>work_mutex: Threads::Mutex; work_mutex: Threads::Mutex; work_per_worker: int: // protected by work_mutex work_per_worker: int: // protected by work_mutex work_per_worker: int: // protected by work_mutex Warker: type = interface inherits Threads::Thread 3; // an unusual interface: no methods beyond Thread::startup and the other Thread methods WorkImpl: type = implement startup: method (our_workers: in int)</pre>	ThreadedSimulation: module work_mutex: Threads: Mutex; work_mutex: Threads: Mutex; work_per_worker: int; // protected by work_mutex extra_work: int; // protected by work_mutex workper = interface inhents: Threads::Thread: startup and the other Thread methods: // an unusual interface: no methods beyond Thread::startup and the other Thread methods: // an unusual interface: no methods beyond Thread::startup and the other Thread methods: // an unusual interface: no methods beyond Thread::startup and the other Thread methods: // an unusual interface: no methods beyond Thread::startup and the other Thread methods: // an unusual interface: no methods beyond Thread::startup and the other Thread methods: // an unusual interface: no methods beyond Thread::startup and the other Thread methods: // an unusual interface: no methods beyond Thread::startup and the other Thread methods: // an unusual interface: no methods beyond Thread::startup and the other Thread methods: // an unusual interface: no methods beyond Thread::startup and the other Thread methods: // an unusual interface: no methods beyond Thread::startup and the other Thread methods: // an unusual interface: no methods beyond Thread::startup and the other Thread methods: // an unusual interface: no methods beyond Thread::started; delegates most of its work to forked staing workers // and our_work = 1, extra_work = 1; // an work = 1;	delegated to workers: int = (our workers - 1);
<pre> delegated_to_workers: if (delegated_to_workers: if (delegated_to_workers = deleg if (delegated_to_workers = deleg if (left_workers = int = 0; if (left_workers = int = 0; if the following declarations are pr do_work: int = 0; if the workers = int = 0; if the workers = int = 0; if the following : Worker; if eft_sibing: Worker; if eft_sibing:</pre>	<pre> delegated_to_workers: if if (delegated_to_workers: if if (delegated_to_workers: a deleg if (morkers: a deleg if (left_workers: int = 0; ight_workers: int = 0; if the following declarations are pr do_work(our_work); if the following declarations are pr do_work(our_work); if the following declarations are pr if the</pre>	<pre>} } delegated_to_workers.if (delegated_to_workers.if if(delegated_to_workers = deleg if(teft_workers = deleg if(teft_work</pre>	<pre>delegated_to_workers: int = (our_workers - 1); if (delegated_to_workers > 0) { if (delegated_to_workers - 1); if (delegated_to_workers.'); if (delegated_to_workers.'); if (delegated_to_workers.'); if (left_workers > 0) left_sibiling = new Workerinpi(left_workers); if (left_workers = 0) left_workers = left_workers); if (left_workers = 0) left_sibiling = new Workerinpi(left_workers); if (left_workers = 0) left_workers = left_workers); if (left_workers = 0) left_workers = left_workers = left_workers; if (left_workers = 0) left_workers = left_workers = left_workers = not = 0; if (left_workers = 0) left_workers = not = 0; if (left_workers = 0) left_workers = not = 0; if (left_workers = 0) left_workers = not = 0; if (left_workers = 0) left_workers = not = 0; if (left_workers = 0) left_workers = not = 0; if (left_workers = 0) left_workers = not = 0; if (left_workers = 0) left_workers = not = 0; if (left_workers = 0) left_workers = not = 0; if (left_workers = 0) left_workers = not = 0; if (left_workers = 0) left_workers = not = 0; if (left_workers = 0) left_workers = not = 0; if (left_workers = 0) left_workers = not = 0; if (left_workers = 0) left_workers = not = 0; if (left_workers = 0) left_workers = not = 0; if (left_workers = 0) left_workers = not = 0; if (left_workers = 0</pre>	<pre>delegated_lo_workers.int = (our_workers 1); if (delegated_lo_workers > 2); { if (delegated_to_workers > 2); { if (delegated_to_workers > 0); if (delegated_to_workers); if (left_workers = delegated_to_workers); if (left_workers = delegated_to_workers); if (left_workers > 0); if (left_workers); if (left_workers); lift to (left_workers); lift workers; int = 0; lift to (left workers; int = 0; lift workers; int = 0; lift to (left to (left workers; int = 0; lift to (left to (left workers; int = 0; lift to (left workers; int = 0; lift to (left to (left workers; int = 0; lift to (left to (left workers; int = 0; lift to (left workers; int = 0; lift to (left to (left workers;</pre>	<pre>} } delegated_to_workers int = (our_workers - 1); if (delegated_to_workers > 0) { if (delegated_to_workers > 0) { if (delegated_to_workers = delegated_to_workers)^; if (norkers = delegated_to_workers)^; if (left_workers = delegated_to_workers)^; if (left_workers > 0) left_stbing = new Workerimpt(left_workers); if (left_workers > 0) left_stbing = new Workerimpt(nght_workers); if (left_works) = new Workerimpt(nght_workers); if (norkers); if (left_workers > 0) left_stbing = new Workerimpt(nght_workers); if (left_works); if (left_workers > 0) left_stbing = new Workerimpt(nght_workers); if (left_works); if (left_workers > 0) left_stbing = new Workerimpt(nght_workers); if (left_works); if (left_workers > 0) left_stbing = new Workerimpt(nght_workers); if (left_works); if (left_workers > 0) left_stbing = new Workerimpt(nght_workers); if (left_works); if (left_works); if (left_workers > 0) left_stbing = new Workerimpt(nght_workers); if (left_works); if (left_works)</pre>	<pre>delegated_to_workers int = (our_workers - 1); if (delegated_to_workers > 0) { if (delegated_to_workers = delegated_to_workers); if (int_workers = delegated_to_workers); if (int_workers); if (int_workers); if (int_workers); if (int_workers); if (int_workers); if (int_workers); if do_work(our_work); if do_work(our_wo</pre>	<pre> } delegated_to_workers: int = (our_workers - 1); if (delegated_to_workers > 2) { if (delegated_to_workers > 2) { if (delegated_to_workers > 2) { if (defl_workers = delegated_to_workers); if (left_workers = delegated_to_workers); if (left_workers > 0) left_stbling = new Workerimpl(left_workers); if (left_workers > 0) left_stbling = new Workerimpl(injht_workers); if the following declarations are private to the WorkerImpl implementation // the following declarations are private to the WorkerImpl implementation </pre>		<pre>vorkinp: type = impenentation of violiner { inplement startup: method (our_workers: in int) // in int) // in executed when the thread is started; delegates most of its work to forked sibling workers within work_mutex { // acquire work_mutex for duration of the within statement our_work = work per_worker; // acquire work_mutex for duration of the within statement // in the formation of the mithin statement // in the formation of the mithin statement // in the more for mithin work_mutex for duration of the within statement // in the more for mithin statement // in the mithin statement // in the more for more for mithin statement // in the more for mithin statement // in the more for mithin statement // in the more for more for mithin statement // in the more for more for mithin statement // in the more for mithin statement // in the more for more for mithin statement // in the more for more for mithin statement // in the more for more for mithin statement // interval for mithinterval for mithin</pre>	<pre>work_mutex: Threads:.Mutex; work_mutex: Threads:.Mutex; work_per_works:.nt;</pre>	ThreadedSimulation: module work_mutex: Threads::Mutex; work_per_work: int; // protected by work_mutex work_per_work: int; // protected by work_mutex work_met type = interface inherits: Threads::Thread B; // an unusuel interface: no methods beyond Thread::tathup and the other Thread methods workImpl: type = implementation of Worker implement startup: method (our_workers: in int) // implement the thread is started; delegates most of its work to forker shing workers within work_mutex // acquire work_mutex for duration of the within statement our_work = work_per_workers: in int)	$\# \{ extra_work > 0 \} \{ our_work > 1 \} extra_work = 1 \} extra_work = 1 \}$
<pre># (extra_work > 0) (0 } delegated_to_workers: # delegated_to_workers: # delegated_to_workers: # if (delegated_to_workers = deleg nght_workers = deleg nght_workers > 0) nght_sibling = new \ } nght_sibling = new \ } nght_sibling = new \ if (def_work (our_work); } nght_sibling = new \ if (def_work (our_work); if (eff_workers: int = 0; if (if work ers: int = 0; if wo</pre>	<pre># (extra_work > 0) (0 } debegsted_to_workers: # debegsted_to_workers: # debegsted_to_workers: # def_workers = deleg nght_workers = deleg nght_workers = deleg nght_workers > 0) nght_sibling = new \ } f(left_workers = b do_work(our_work); } f(left_workers = b do_work(our_work); f(left_workers = b nght_sibling: Worker; f(left_workers = b) nght_sibling: Worker; } </pre>	<pre># (extra_work > 0) ((</pre>	<pre># (extra_work > 0) (our_work += 1; extra_work -= 1;) delegated to workers int = (our_workers - 1); delegated to workers > 0) { if (delegated to workers > 0) { if (delegated to workers > 0) { if (delegated to workers > 0) {</pre>	<pre># (extra_work > 0) (our_work += 1; extra_work -= 1;) delegated_lo_workers: int = (our_workers - 1); if (calegated_lo_workers > 0) { if (calegated_lo_workers > 0) { if (calegated_lo_workers > 0) {</pre>	<pre>% (extra_work > 0) {our_work += 1; extra_work -= 1;} 3 delegated_b_workers int = (our_workers - 1); if (delegated to_workers > 2) { if (delegated to_workers > 2) { if (delegated to_workers = delegated to_workers; if (delegated to_workers = delegated to_workers; if (left_workers > 0) left_sibling = new Workerinpi(left_workers; if (left_workers; int = 0; left_workers > 0) left_sibling = new Workerinpi(left_workers; left_workers; int = 0; left_workers</pre>	<pre># {extra_work > 0} {our_work > 1; extra_work -= 1;} } * (extra_work = 0; {urr_workers: int = (our_workers - 1); if delegated_to_workers: int = (our_workers.); if delegated_to_workers.); if (left_workers = delegated_to_workers); if (left_workers = delegated_torkers); if (left_workers); if (left_workers = delegated_torkers); if (left_workers); if (</pre>	<pre># (extra_work > 0) (our_work += 1; extra_work -= 1;) # (extra_work = 1; extra_work -= 1;) # delegated to_workers : nt = (our_workers - 1); # (delegated to_workers > 0) {</pre>	~9 9 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	<pre>working: type = impendation of workers { inplement startup: method (our_workers: in int)</pre>	<pre>work_mutex: Threads::Mutex; work_mutex: Threads::Mutex; work_int;</pre>	ThreadedSimulation: module work_mutex: Threads:.Mutex; work_per_worker: int; // protected by work_mutex work_per_work: int; // protected by work_mutex work_int; // in nusual interface inherits Threads::Thread B; // an unusual interface: no methods beyond Thread::startup and the other Thread methods // interment startup: method (our_workers: in int) // executed when the thread is started; delegates most of its work to forked sitking workers within work mutex // // aronia work mutex // // aronia work mute interent	ğ
our work = work p if (extra_work = work p detegated_to_workers: i detegated_to_workers: i if (delegated_to_workers: a deleg nght_workers = deleg nght_sibling = new } do_work(our_work); = deleg if (left_workers = deleg nght_sibling = new } if (left_workers = deleg if (left_workers = deleg i	our_work = work_p if (extra_work = work_p) { detegated_to_workers: 1 detegated_to_workers: 1 if (detegated_to_workers: 2 if (detegated_to_workers: 2 if (left_workers: 4 if (left_workers: 4 do_work(our_work); if (left_workers: 4 do_work(our_work); if (left_workers: 4 do_work(our_work); if (left_workers: 4 do_work(our_work); if (left_workers: 4 if (left_workers: 4 if (left_workers: 4 if (left_workers: 4 if (left_worker; 4 if (left_workers: 4 i	our work = work p if (extra_work > 0) { f (extra_workers:) delegated_to_workers:) if (delegated_to_workers:) if (left_workers = deleg right_workers = deleg right_workers = deleg f (left_workers = deleg if (left_workers = deleg right_workers: int = 0; if (left_workers: int = 0; if (work ers: int = 0; if morkers: int = 0; if abiling: Worker; right_sibling: Worker;	<pre>our_work = work_per_worker; # (extra_work > D) (our_work == 1; extra_work == 1; # (extra_work > D) (our_workers - 1); # (delegated_to_workers > 2) { # (workers = delegated_to_workers); # (workers = delegated_to_workers); # (ieft_workers > 0) ieft_soling = new Workerimpi(ieft_workers); # (ieft_workers > 0) ieft_soling = new Workerimpi(inft_workers); # norkers int = 0; # the toilowing declarations are private to the Workerimpi implementation do_work int = 0; if the toilowing Worker; int = 0; ight_workers: int = 0; ight_workers: int = 0; ight_workers: int = 0; if notif delegates, manages "ieft_workers" workers if notif delegates, manages "ieft_workers" workers if notif inter Workers" inter 0; if notif delegates, manages "ieft_workers" workers if notif inter Workers" inter 0; if notif inter Workers" inter 0; if notif i</pre>	<pre>our_work = work_per_worker; # (extra_work = 0) (our_workers: art = (our_workers = 1;) delegated_to_workers: art = (our_workers = 1); delegated_to_workers = delegated_to_workers.); if (delegated_to_workers = delegated_to_workers.); if (left_workers = delegated_to_workers.); if (left_workers.); if (l</pre>	<pre>our_work = work per_worker;</pre>	<pre>our_work = work_per_worker;</pre>	<pre>our_work = work_per_worker;</pre>	our_work = work_p if (extra_work > 0) { delegated_to_workers: i delegated_to_workers: i if (delegated_to_workers: dele if (left_workers = del	<pre>**orkinps: type = implement or vionker { inplement startup: method (our_workers: in int)</pre>	<pre>work_mutex: Threads::Mutex, work_mutex: Threads::Mutex, work_per_worker: int;</pre>	ThreadedSimulation: module work_mutex: Threads: Mutex; work_per_work:::nt; // protected by work_mutex work_per_type = interface inherits Threads::Thread B; Warker: type = interface: no methods beyond Thread::startup and the other Thread nethods WorkImpl: type = implement startup: method (our_workers: in int); implement startup: method (our_workers: in int);	1 in executed when up and an executed when up and an and, up and an
ດ ເຊິ່ງ ເລີ່ ເຊີ່ ເຊີ່ ເຊີ່ ເລີ່ ເລີ່ ເລີ ເລີ່ ເລີ່ ເລີ່ ເລີ ເລີ ເລີ່ ເລີ ເລີ ເລີ ເລີ ເລີ ເລີ ເລີ ເລີ ເລີ ເລີ	ំ ស្រុក ស ស ស្រុក ស្រុក ស្រុក ស្រុក ស្រុក ស្រុក ស ស ស ស ស ស ស ស ស ស ស ស ស ស ស ស ស ស ស	ំដែរ ស្រុក សំរុក សំរាក សំរុក សំរុក សំរុក សំរុក សំរុក សំរុក សំរុក សំរុក សំរុក សំរាក	<pre>writin work_muter {</pre>	<pre>within work, mutex {acquarkers,acquarkers,acquarkers,acquarkers,acquarkers,acquarkers,acquarkers,acduarkers, and within work,acduarkers, int = (ourwork + = t; extrawork = t,);</pre>	<pre>within work mutax {</pre>	<pre>within work_mutes { // accurate work_mutes for duration of the within statement</pre>	<pre>within work mutex { // accurate work mutex for duration of the within statement within work = work per_worker; # (extra_work > 0) (our_work = 1; extra_work = 1;) delegated_to_workers: int = (our_workers - 1); if delegated_to_workers = delegated_to_workers.); if delegated_to_workers = delegated_to_workers - i; if (left_workers = delegated_to_workers.); if def_workers = delegated_to_workers.); if def_workers > 0) [act_workers - ibft_workers); if def_workers = delegated_to_workers.); if def_workers = delegated_to_workers.]; if def_workers = delegated_toworke</pre>		$\frac{1}{2}$	<pre>work_mutex: Threads:.Mutex; work_mutex: Threads:.Mutex; work_per_worksr: int; // protected by work_mutex extra_work: int; // protected by work_mutex Warker: type = interface inherits Threads:.Thread by work_mutex // an unusual interface: no methods beyond Thread.:startup and the other Thread methods WorkImpi: type = implementation of Worker {</pre>	ThreadedSimulation: module work_mutex: Threads: Mulex; work_mutex: Threads: Mulex; work_mutex; work_mutex; work_mutex; Worker: type = interface inherits: Threads:: Thread by work_mutex Workimpi: type = implementation of Worker; Workimpi: type = implementation of Worker	implement startup: method (ω r_workers: in int) I=N everyted when the thread is strated references of its work to forked sition undrare
inpleme inplem	inpleme inpleme fits for for soft_sibilities for fight_work	inpleme (<pre>intplement starture. method (our workers: in int) {</pre>	<pre>implement startup: method (our_workers: in int)</pre>	<pre>implement stantury: method (our_workers: in int)</pre>	<pre>implement startup: method (our_workers: in int) {</pre>	<pre>implement startup: method (our_workers: in int)</pre>	<pre>inplement startup. method (our_workers: in int) {</pre>		<pre>work_mutex: Threads::Mutex; work_mutex: Threads::Mutex; work_per_worker: int;</pre>	ThreadedSimulation: module work_mutex: Threads: Mutex; work_per_worker: int: // protected by work_mutex extra_work: int; // protected by work_mutex Warker: type = interface inherits Threads:: Thread by work_mutex // an unusual interface: no methods beyond Thread::startup and the other Thread nethods	
Workimpi: typ implement i implement i i i i i i i i i i i i i i i i i i i	Workimpi: typ implement i implement i i i i i i i i i i i i i i i i i i i	orkimpi: tyr impleme { / / / / / / / / / / / / / / / / / /	<pre>workImpl: type = implementation of Worker: implement startup: method (our_workers: in int)</pre>	<pre>WorkImpt: type = implementation of Worker implement statuto: method (our_workers: in int) implement statuto: method (our_workers: in int) implement statuto: method (our_workers: in int) implement within work.muter k1 accute work.pet.worker:</pre>	<pre>WorkImpl: type = implementation of Worker implement startup: method (cur_workers: in int)</pre>	<pre>WorkImpl: type = implementation of Worker implement startup: method (our_workers: in int) implement startup: method (our_workers: in int) i</pre>	<pre>Workimp: type = implementation of Worker { inplement startup: method (our_workers: in int) (inplement startup: method (our_workers: in int) (interview when the stread is started; delegates most of its work to forked stilling workers within work_mutex { if acquire work_mutex for duration of the within statement work > 0) (our_work + 1; extra_work = 1;) if (delegated_lo_workers: nf = (our_workers - 1); if (delegated_lo_workers > 0) is f_sibling = new WorkerImp(ifght_workers); ight_sibling = new WorkerImp(ifght_workers); ight_sibling = new WorkerImp(ifght_workers); ight_sibling = new WorkerImp(ifght_workers); if (left_workers = 0) is f_sibling = new WorkerImp(ifght_workers); ight_sibling = new WorkerImp(ifght_workers); if (left_workers = 0) is f_sibling = new WorkerImp(ifght_workers); if (left_workers = 0) is f_sibling = new WorkerImp(ifght_workers); if (left_workers = 0) is f_sibling = new WorkerImp(ifght_workers); if (left_workers = 0) is f_sibling = new WorkerImp(ifght_workers); if (left_workers = 0) is f_sibling = new WorkerImp(ifght_workers); if (left_workers = 0) is f_sibling = new WorkerImp(ifght_workers); if (left_workers = 0) is f_sibling = new WorkerImp(ifght_workers); if (left_workers = 0) is f_sibling = new WorkerImp(ifght_workers); if (left_workers); if (left</pre>	<pre>WorkImpl: type = implementation of Worker inplement startup: method (our_workers: in int) inplement startup: method (our_workers: in int) i</pre>		. Mutex,	iute x;	Warker: type = interface inherits Threads::Thread 8; // an unusual interface: no methods beyond Thread::startup and the other Thread methods
Warker: type Marker: type Morkinpi: type inplement	Warker: type // an unu // an unu // an unu // an unu // implemer // implemerer // implemerer /	arker: type // an unu // an unu // an unu // an // an unu // // // // // // // // // // // // //	<pre>Worker: type = interface interface: no methods heyond Thread::startup and the other Thread are</pre>	<pre>Worker: type = interface intents: Thread 0;</pre>	<pre>Worker: type = interface inherits. Threads:: Thread::istrup and the other Thread methods // an unusual interface. no methods heyond Thread::istrup and the other Thread methods // an unusual interface. no methods heyond Thread::istrup and the other Thread methods // an unusual interface. no methods heyond Thread::istrup and the other Thread methods // an unusual interface. no methods heyond Thread::istrup and the other Thread methods // an unusual interface. no methods heyond Thread::istrup and the other Thread methods // an unusual interface. no methods heyond Thread::istrup and the other Thread is intit // a excuted when the thread is statted; delegates most of its work to brind // accurate work</pre>	<pre>Worker: type = interface interface: no methods beyond Thread::startup and the other Thread methods</pre>	<pre>Worker: type = interface interface interface interface: no methods Peyond Thread::startup and the other Thread methods // an unusual interface: no methods Peyond Thread::startup and the other Thread methods Workmpi: type = implementation of Worker // an unusual interface: no methods Peyond Thread::startup and the other Thread methods // an unusual interface: no methods Peyond Thread::startup and the other Thread // an unusual interface: no methods Peyond Thread::startup and the other Thread // an unusual interface: no methods Peyond Thread::startup and the other Thread // an unusual interface: no methods Peyond Thread::startup and the other Thread // an unusual interface: no methods Peyond Thread::startup and the other // acquire work_mutex { // acquire work_mutex for duration of the within statement // acquire work = work_per_worker: = (aur_work = 1; atra_work = 1; atra_work = 1; atra_work = 1; // acquire workers = delegated_for_workers = delegated_for_workers = delegated_for_workers = idplit_workers = idplit_workers = delegated_for_workers = idplit_workers = idplit_morkers = if workers = idplit_morkers = idplited = new Workers = idplited = new Wo</pre>	<pre>Warker: type = interface interface interface in methods beyond Thread::ttertup and the other Thread methods</pre>	Worker: type = interface inherits Threads::Thread 3; // an unusual interface: no methods beyond Thread::startup and the other Thread methods	ಿಗೊtex;	itex.	
extra_work: it Warken: type // an unu // an unu // impleme // // the folk // the folk // fite folk // work/ // fite folk	extra_work: itype Worksm: type Worktmph: typ inpleme: inp	tra_work: it arken: type implement implement implement implement implement in the foll is the work is the work is the work is the work	<pre>extra_work: art,</pre>	<pre>extra_work: int.</pre>	<pre>exta_work: int,</pre>	<pre>extra_work: int, // protected by work_mutex. Warker: type = interface. no methods beyond Thread::startup and the other Thread methods workimpi: type = implementation of Worker: in int) if an unusual interface. no method (aur_workers: in int) if interesting method (aur_workers: in int) if interesting method (aur_workers: in int) if interesting method (aur_workers: int) if interesting method (aur_workers: int) if interesting method (aur_workers: int) if (athe_workers: int = (our_workers: -1); if (athe_workers: int = (our_workers: -1); if (athe_workers: = aen workers: -1); if (athe_workers: = allogated_1), workers: -1); if (athe_workers: = arentod (work_r); if (athe_workers: = and workers: -1); if (athe_workers: = arentod (workers: -1); if (athe_workers: = arentod (work_r); if (athe_workers: = arentod (work1); if (athe_workers: = arentod (work_r); if (</pre>	<pre>exta_vont: int,</pre>	<pre>extra_work: int; // protected by work_mutex Worker: type = interface interface. Thread::tertup and the other Thread interface Workingh: type = implementation of Worker Workingh: type = implementation of Worker implement startup: method (our_workers: int) implement startup: int) implement startup: method (our_workers: int) implement startup: int) implement startup: method (our_workers: int) implement startup: int impl</pre>	extra_work: int; // protected by work_mutex Worker: type = interface inherits Threads::Thread 0; // an unusual interface: no methods beyond Thread::startup and the other Thread methods	have	ThreadedSimulation: module {	ಿಗೆರ್ಚಲ್ಲು
work_mutak: work_per_work: // an unu // an unuu // an unuuu // an unuuuu // an unuuuu // an unuuuuuu	work_munks:: work_per_work: // an unu // an unuu // an unuu // an unuu // an unuu // an unuu //	ax mutax: arker: type arker: type an unu: an	<pre>wook_index: Interacts Mate, wook_index: Thereader III;</pre>	<pre>work_metak.streads.th. Matex, work_pee_works.int, // protected by work_mutex extra_work.int, // protected by work_mutex Monker: type = interface: no methods beyond Thread::sturtup and the other Thread methods Monker: type = implement startup: method (sur_workers: in int) // a unusuel interface: no methods beyond Thread::sturtup and the other Thread methods Monking: type = implement startup: method (sur_workers: in int) // method (sur_workers: in int) // method (sur_workers: int) // feeled of work, per work, mutex for duration of the within statement our_work = work_per_workers: int // workers: if // workers: // feeled of workers = delegated_to_workers: if // workers: // feelegated_to_workers = delegated_to_workers: if // workers; // feelegated_to_workers = delegated_to_workers: // feelegated_to_workers = othegated_to_workers: // feelegated_to_workers = othegated_to_workers: // fielef_workers = othegated_to_s = int) // fielefed_to_s = int) // fielefedeatedatedatedatedatedatedatedatedateda</pre>	<pre>wow_more intracests haves, wow_intrements intreasts haves, introduction, wowing interfaces interfaces interfaces haven interfaces introduce and worker type = interfaces interfaces. In the interface interfaces interfaces in the implement startup: method (our_workers: in int) {</pre>	<pre>work_matrix means. means. means. work_mert. The ans. means. means. work_mert. work_mint. // protected by work_mutex crtra_work: int. // protected by work_mutex means the other Thread B:</pre>	<pre>work_mnuext: intexact. intex work_mutext: int;</pre>	<pre>wow.mode::intex.:intex.: // protected by work_mutex. wow.per_work=:int, // protected by work_mutex work.per_work=:int, // protected by work_mutex Workmpt::type = interface: inherits Threads::Thread::Th</pre>	<pre>work_mutax: inreads.int;</pre>		ThreadedSimulation: module	

Lewis in view of Dyer and further in view of Proebsting	up: method (our_workers: in int) // executed when the thread is started; delegates most of its work to forked sibling worklers within work_mutex { // acquire work_mutex for duration of the within statement	work_mutex	Threadhenter		work_per_worker	our_work		extra_work	integer constant \mathcal{C}_i type \mathcal{D}	extra_work>0			our_work			integer constant 1, type 2	our_work+=1	our_work		extra_work		intener constant 1 tune 3		extra work	if(extra_work≻0)(
Lewis in view of Dye	<pre>startup: method (our_workers: in ini) {</pre>	LosdGloba(2)	invokeOuter(0x000810004)	our_work = work_per_worker;	LoadGlobal(1)	Store Global(5)	if $(extra_work > 0)$ (LoadGlobai(3)	Loadint(0, 0)	Compareint(>)	SkipThen(cond_false_tag_2)	0.005 ± 1	GiobalAddr(5)	Dup()	Loadindirect(0)	Loadint(0, 1)	Addint()	Store indirect(D)	eztra_work -= 1;	GlobalAddr(3)	Euse() to and entire of 000	LoadSinect(0 1)	Subiation	Storeindirecti0)	SkipEise(end_if_tag_3)	BeginElse(cond_false_tag_2)	,×***,
U.S. Patent No. 6,061,520 - Claim 7																											

U.S. Patent No. 6,061,520 - Claim 7	Lewis in view of Dyer s	Lewis in view of Dyer and further in view of Proebsting
	LoadGlobai(0) invokeOutser(040305)	work_mutex Tkread∵esti
	delegated to workers: int = (our workers - 1);	
	LoadArg(D)	
	LOBGINUU, 3) Souther	Niteger constant 1, type 2 Aur avorbere_3
	StoreLocal(2)	delegated_to_workers
	u. do work(our work);	
	LosaGlotas(*)	do_wark
	LoadGlokak(5)	strow_such
	ProcCall(7)	CO_WORK(OUT_NORK)
	ProcReturn(1)	method(our_workers
	Figure 6: MCode instructions generate	Figure 6: MCode instructions generated for the μC^{++} test program's startup method
	Lewis at 124-125, Figs. 5-6.	
	To the extent <i>Lewis</i> does not explicitly diskill in the art would combine the disclosi	To the extent <i>Lewis</i> does not explicitly disclose this limitation of Claim 7, one of ordinary skill in the art would combine the disclosure of <i>Lewis</i> with <i>Dyer</i> to arrive at the claimed
	feature. Specifically, <i>Dyer</i> shows that an instruction for initializing a data struc array) was widely known in the art, as evidenced by its discussion of static array initialization:	in instruction for initializing a data structure (e.g., an idenced by its discussion of static array
	"Mocha transformed a static initializer into an elegant, but illegal, construction:	to an elegant, but illegal, construction:
	public ConsoleWindow(String str	string, int il)
	{ dead = false; styles = { "Plain", "Bold", "Italic" sizes = { "8", "9", "10", "12", "14",	"Italic" }; 12", "14", "16", "18", "24" };
	Bracketed initializer lists for arrays are ve (either class or local), not for other assign to me, but I'm sure Sun must have had a	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these

U.S. Patent No. 6,061,520 - Claim 7	Lewis in view of Dyer and further in view of Proebsting
	initializers are actually implemented by inline code inside constructors, generated by the compiler.
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." <i>Dyer</i> at 3.
wherein the play executing step includes the step of: play executing the code to identify the initialization of the data structure.	<i>Lewis</i> discloses wherein the play executing step includes the step of: play executing the code to identify the initialization of the data structure. For example, <i>Lewis</i> discloses simulating execution of the MCode instructions in the MCode object files, and that the MCode instruction set includes instructions for statically initializing data structures.
	"The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." <i>Lewis</i> at 126.
	"In order to do high-quality code generation, we need to rebuild the expression trees from the stack machine. The code generator defers generation until the final target for an expression is known. Much of the process is similar to that of TNBIND algorithm documented in [Wulf 75]. This algorithm gives excellent results and executes extremely efficiently. A final optimization does instruction reordering to minimize RISC processor pipeline execution conflicts. By organizing the code generator into a series of cascading object streams, we are able to consume MCode and generate native machine code in one pass. Our object-oriented architecture provides an efficient way to trade increased memory for speed." <i>Lewis</i> at 127.
	To the extent <i>Lewis</i> does not explicitly disclose this limitation, one of ordinary skill would combine <i>Lewis</i> with <i>Dyer</i> and <i>Proebsting</i> to arrive at the claimed feature. Specifically, <i>Dyer</i> and <i>Proebsting</i> together show that the play execution of byte codes against a memory to identify underlying code, such as the static initialization of an array, was commonly

U.S. Patent No. 6,061,520 - Claim 7	Lewis in view of Dyer and further in view of Proebsting
	known to those of ordinary skill in the art at the time of the invention. For example, the decompiler technology disclosed by <i>Proebsting</i> discloses a technique for decompilation of Java byte code that play executes the Java virtual machine's stack:
	"This paper presents our technique for automatically decompiling Java bytecode into Java source." <i>Proebsting</i> at Abstract.
	" <i>Decompilation</i> transforms a low-level language into a high-level language. The Java Virtual machine (JVM) specifies a low-level bytecode language for a stack-based machine. This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a <i>class file</i> that contains type information and bytecode. The JVM requires a significant amount of type information from the class files for object linking Decompilation systems can exploit this type of information and well-behaved property to recover Java source code from the class file.
	We present a technique for transforming low-level Java bytecode into legal Java source code." <i>Proebsting</i> § 1.
	"We have implemented a prototype Java decompiler, Krakatoa, in Java." Proebsting § 5.
	"Krakatoa uses a stack simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations." <i>Proebsting</i> § 1.
	"Symbolic execution of the bytecode creates the corresponding Java source expressions Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." <i>Proebsting</i> § 2.
	Thus it would have been obvious to one of ordinary skill in the art to combine the methods of <i>Lewis</i> with the play execution (symbolic execution) of bytecode against a memory, as disclosed by <i>Proebsting</i> . And, even more specifically, the artisan would look to known

U.S. Patent No. 6,061,520 – Claim 7	Lewis in view of Dyer and further in view of Proebsting
	decompilers, such as those disclosed in <i>Dyer</i> . <i>Dyer</i> reviews three popular decompilers that were available before the '520 patent was filed: DejaVu, Mocha, and WingDis. <i>See Dyer</i> at 1. In that review, <i>Dyer</i> analyzes the decompilation of code, including the decompilation of static initializers:
	"Mocha transformed a static initializer into an elegant, but illegal, construction:
	public ConsoleWindow(String string, int il)
	<pre>{ dead = false; dead = false; styles = { "Plain", "Bold", "Italic" }; sizes = { "8", "9", "10", "12", "14", "16", "18", "24" }; </pre>
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these
	initializers are actually implemented by inline code inside constructors, generated by the compiler.
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." $Dyer$ at 3.
U.S. Patent No. 6,061,520 – Claim 8	<i>Lewis</i> in view of <i>Dyer</i> and further in view of <i>Proebsting</i>
8. The method of claim 6 wherein the operation statically initializes an array and	<i>Lewis</i> discloses the method of claim 6 wherein the operation statically initializes an array. For example, <i>Lewis</i> discloses that the MCode instruction set includes instructions for statically initializing an array, e.g., "AllocArray."
	"Although MCode includes instructions and data structures needed to implement some Clarity language-specific constructs such as its exceptions and method calls, the core of MCode is suitable for representing code for C and many other languages." <i>Lewis</i> at 119.

U.S. Patent No. 6,061,520 - Claim 8	Lewis in view of Dyer and further in view of Proebsting	ag
	Allow A second more than a long only. Allow A second more than a long only	d consult months
		ATAC UNDER State Association
	ArrayLength(array type index)	ase(int)
	BeginDefaultCase()	iCase()
	InvokeOuter(method type index) EndCase(end switch	icí swítch
	invokeDe/egated(method type index)	
	Widen(base interface number) DoBreak(end tag)	d (ag)
	Narrow(target type index)	(end tag)
	Figure 4: The MCode instruction set	ar set
	Lewis at 123 Fig. 4 (excerpt).	
	"Figures 5 and 6 give an example of generated MCode. The Clarity method <i>startup</i> in Figure 5 produces the MCode instructions shown in part in Figure 6."	hod <i>startup</i> in

ThestedShrunshor: module work_matter. Threade: Muck: work_matter. Threade: Muck: work_matter. Threade: Muck: work_matter. prove interface interface interface interface interface. Workmap: type = implementation of Work_matter an unual interface. no methods beyond. Thread. Thread methods Workmap: type = implementation of Work interface. Interface interface interface. Workmap: type = implementation of Work interface. Interface interface interface. Workmap: the implementation of Work interface. Interface interface interface. Workmap: the implementation of the work to forked siling workers when work interface. Interface more interface. Workmap: the implementation of the work in the work to forked siling workers interface. Interface more interface interface. Workmap: the implementation of the work in the work in the work in the interface. Workmap: the implementation of the work in the wo	U.S. Patent No. 6,061,520 - Claim 8	Lewis in view of Dyer and further in view of Proebsting
work_mutax: work_per_wo kan unu Worker: type // an unu // an unuu // an unuu // an unuu // an un		ThreadedSimulaiion: module
Warker: type // an unu // an unu // an unu // inpleme // inpleme // inpleme // inpleme // inpleme // inpleme // inpleme // inpleme // inpleme		. M utex,
Workimpht tyr impleme impleme impleme impleme impleme impleme impleme impleme impleme impleme impleme impleme impleme		Worker: type = interface inherits. Threads::Thread 0; // an unusual interface: no methods beyond Thread::startup and the other Thread methods
ingleme ato work is right wood is the foll is the wood is the site		Workimpi: type = implementation of Worker
<pre>if (delegated_to_workers: if if (delegated_to_workers: deleg ieft_workers = deleg ieft_workers = deleg if (left_workers = deleg if (left_workers > 0) inght_sibling = new \ inght_sibling = new \ do_work(our_work); if if the following declarations are pr do_work: int = 0; ieft_workers: int = 0; ieft_workers: int = 0; ieft_workers: int = 0; ieft_workers: int = 0; ieft_sibling: Worker; inght_workers: int = 0; ieft_sibling: Worker; inght_sibling: Worker;</pre>		<pre>inplement startup: method (our_workers: in int) {</pre>
<pre>} do_work(our_work); } do_work: method (work_to_do: in do_work: int = 0; ieft_workere: int = 0; ieft_stiting: Worker; indent_stiting: Worker; } Figure 5: Part of</pre>		ر detegated_to_workers: int = (our_workers - 1); ieft_workers = detegated_to_workers/2; nght_workers = detegated_to_workers/2; if (teft_workers = detegated_to_workers - teft_workers; if (teft_workers > 0) teft_stbling = new Workertimpi(teft_workers); nght_stbling = new Workertimplitight_workers);
<pre>// the following declarations are pr do_work: method (work_to_do: in cur_work: int = 0; iseft_workers: int = 2; isght_workers: int = 2; isght_sibling: Worker; isght_sibling: Worker; </pre>		ده ر
our_work: int = 0; ieft_workers: int = 2; right_workers: int = 2; ieft_sitting: Worker; right_sitting: Worker; Figure 5: Part of		h the following declarations are private to the WorkerImpi implementation do_work: method (work_fo_do: in int) { f estided $*f$ }.
az 8		อเน_พอกk: int = 0; left_workers: int = ช; right_workers: int = ช; ieft_sibing: Worker; right_sibing: Worker;
Figure 5: Part of the Clarity version of the μC^{++} test program		
		Figure 5: Part of the Clarity version of the μC^{++} test program

Lewis in view of Dyer and further in view of Proebsting	iup: method (our_workers: in int) // executed when the thread is started; delegates most of its work to forked sitking worklers within work_mutex { // acquire work_antlex for duration of the within statement	work_mutex	Threather	-	work per worker our work	1	extra_work	integer constant 0, type 0	extra_work>0			our_work			integer constant 1, type 0	our_work+=1	our_work		extra_work		integer constant 1, type 0	extra_work-=1	extra_work		(2)	
Lewis in view of D	startup: method (our_workers: in init) { // executed when the thread is started; d within work_mutex { // acquire work_	LosdGlobat(2)	invokeOuter(0x02010204)	our_work = work_per_worker;	Lasatslotsai(1) StoreGiobai(5)	if (extra_work > 0) (LoadGlobai(3)	Losdint(0, 0)	Compareint(>)	SkipThen(cond_faise_tag_2)	$our_work += 1;$	GiobalAddr(5)	Dup()	Loadindirect(0)	Loadint(0, 3)	Addint()	StoreIndirect(D)	extra_work -= 1;	Giotral Addr(3)	Dup() Loadindirect(0)	LoadSigned(0, 1)	Subint()	StoreIndirect(U)	SkipEise(end_if_tag_3)	BeginElse(cond_false_tag_2) Environment # toon at	ž
U.S. Patent No. 6,061,520 - Claim 8																										

U.S. Patent No. 6,061,520 - Claim 8	Lewis in view of Dyer 2	Lewis in view of Dyer and further in view of Proebsting
	EcadGlobai(0)	work_mutex Thread-text
	delegated_to_workers: int = {our_workers - 1);	
	LoadArg(0)	
	LOBGINU, 3) Sector #1	integer constant 1, type 2
	StoreLocal(0)	detegated_to_workers
	 do work(our work):	
		do_work
	LoadGlobai(5)	show_sho
		co_work(our_work)
	EsecReturn(1)	method(our_workers
	Figure 6. MCode instructions generate	Figure 6: MCode instructions generated for the μC^{++} test program's startup method
	Lewis at 124-125, Figs. 5-6.	
	To the extent <i>Lewis</i> does not explicitly diskill in the art would combine the disclosion	To the extent <i>Lewis</i> does not explicitly disclose this limitation of Claim 8, one of ordinary skill in the art would combine the disclosure of <i>Lewis</i> with <i>Dyer</i> to arrive at the claimed
	teature. Specifically, $Dyer$ shows that an instruction for initializing a data struc array) was widely known in the art, as evidenced by its discussion of static array initialization:	un instruction for initializing a data structure (e.g., an idenced by its discussion of static array
	-	
	"Mocha transformed a static initializer into an elegant, but illegal, construction:	nto an elegant, but illegal, construction:
	public ConsoleWindow(String str	string, int i1)
	<pre>dead = false; dead = false; styles = { "Plain", "Bold", "Italic" sizes = { "8" "9" "10" "12" "14"</pre>	"Italic" }; 12" "14" "16" "18" "24" };
	Bracketed initializer lists for arrays are ve (either class or local), not for other assign to me, but I'm sure Sun must have had a 1	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these

U.S. Patent No. 6,061,520 - Claim 8	Lewis in view of Dyer and further in view of Proebsting
	initializers are actually implemented by inline code inside constructors, generated by the compiler.
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." <i>Dyer</i> at 3.
wherein the play executing step includes the step of: play executing the code to identify the static initialization of the array.	<i>Lewis</i> discloses wherein the play executing step includes the step of: play executing the code to identify the static initialization of the array. For example, <i>Lewis</i> discloses that the MCode instruction set includes instructions for statically initializing an array, e.g., "AllocArray."
	"The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." <i>Lewis</i> at 126.
	"In order to do high-quality code generation, we need to rebuild the expression trees from the stack machine. The code generator defers generation until the final target for an expression is known. Much of the process is similar to that of TNBIND algorithm documented in [Wulf 75]. This algorithm gives excellent results and executes extremely efficiently. A final optimization does instruction reordering to minimize RISC processor pipeline execution conflicts. By organizing the code generator into a series of cascading object streams, we are able to consume MCode and generate native machine code in one pass. Our object-oriented architecture provides an efficient way to trade increased memory for speed." <i>Lewis</i> at 127.
	To the extent <i>Lewis</i> does not explicitly disclose this limitation, one of ordinary skill would combine <i>Lewis</i> with <i>Dyer</i> and <i>Proebsting</i> to arrive at the claimed feature. Specifically, <i>Dyer</i> and <i>Proebsting</i> together show that the play execution of byte codes against a memory to identify underlying code, such as the static initialization of an array, was commonly

U.S. Patent No. 6,061,520 – Claim 8	<i>Lewis</i> in view of <i>Dyer</i> and further in view of <i>Proebsting</i> known to those of ordinary skill in the art at the time of the invention. For example, the
	fo
	"This paper presents our technique for automatically decompiling Java bytecode into Java source." <i>Proebsting</i> at Abstract.
	<i>"Decompilation</i> transforms a low-level language into a high-level language. The Java Virtual machine (JVM) specifies a low-level bytecode language for a stack-based machine. This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a <i>class file</i> that contains type
	information and bytecode. The JVM requires a significant amount of type information from the class files for object linking Decompilation systems can exploit this type of information and well-behaved property to recover Java source code from the class file.
	We present a technique for transforming low-level Java bytecode into legal Java source code." <i>Proebsting</i> § 1.
	"We have implemented a prototype Java decompiler, Krakatoa, in Java." Proebsting § 5.
	"Krakatoa uses a stack simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations." <i>Proebsting</i> § 1.
	"Symbolic execution of the bytecode creates the corresponding Java source expressions Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." <i>Proebsting</i> § 2.
	Thus it would have been obvious to one of ordinary skill in the art to combine the methods of <i>Lewis</i> with the play execution (symbolic execution) of bytecode against a memory, as disclosed by <i>Proebsting</i> . And, even more specifically, the artisan would look to known

U.S. Patent No. 6,061,520 – Claim 8	Lewis in view of Dyer and further in view of Proebsting
	decompilers, such as those disclosed in <i>Dyer</i> . <i>Dyer</i> reviews three popular decompilers that were available before the '520 patent was filed: DejaVu, Mocha, and WingDis. <i>See Dyer</i> at 1. In that review, <i>Dyer</i> analyzes the decompilation of code, including the decompilation of static initializers:
	"Mocha transformed a static initializer into an elegant, but illegal, construction:
	<pre>public ConsoleWindow(String string, int il) {</pre>
	<pre>dead = false; styles = { "Plain", "Bold", "Italic" }; sizes = { "8", "9", "10", "12", "14", "16", "18", "24" }; </pre>
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." <i>Dyer</i> at 3.
U.S. Patent No. 6,061,520 - Claim 9	Lewis in view of Dyer and further in view of Proebsting
9. The method of claim 6 further including the step of: running the created instruction on the processing component to perform the operation.	<i>Lewis</i> discloses the method of claim 6 further including the step of: running the created instruction on the processing component to perform the operation. For example, <i>Lewis</i> discloses that "object files containing MCode are processed by standard linkers or other tools."
	"Runtime generation of machine code offers many advantages. A runtime code generator can take advantage of information about the particular target platform to generate better

U.S. Patent No. 6,061,520 - Claim 10	Lewis in view of Dyer and further in view of Proebsting
	never know whether a called procedure is actually implemented in MCode or in C. (For example, a programmer might have replaced the called procedure using interposition.) This means the interpreter must fully handle all the details required for ABI calls. If a called routine will return an aggregate value, the interpreter must generate a sequence of machine instructions at runtime (a <i>thunk</i>) to support the SPARC ABI's calling convention that the returned aggregate's length must be encoded into a SPARC UNIMP instruction just after the call. The interpreter also stores all program values in memory as SPARC values since this is required for ABI interoperation.
	Recently, a second MCode interpreter has been developed by Mick Jordan. This interpreter executes <i>system models</i> written in the Clarity language. These system models precisely describe how a software system is built: the exact versions of its component parts, all options and build parameters, and how the component parts are assembled. This system modeller is intended to replace the Unix <i>make</i> tool and to eliminate some of its problems: e.g. the inability to exactly reproduce the construction of a software system. The system modeller's MCode interpreter is specialized to executing these models and to interacting with the Clarity program database. It does not need, for example, to support SPARC ABI interoperation." <i>Lewis</i> at 127-28.
	Figure 2: The runtime portion of the Clarity MCode system
	Lewis at 121.

<i>Lewis</i> in view of <i>Dyer</i> and further in view of <i>Proebsting</i> <i>Lewis</i> discloses method of claim 6 wherein the operation has an effect on memory, and wherein the play executing step includes the step of: play executing the code to identify the effect on the memory. For example, <i>Lewis</i> discloses that the concrete subclasses of CGV alue represent the state of the individual entries on the simulated stack, a process which inherently identifies the effect of the play execution on the memory.	"The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGV alue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." <i>Lewis</i> at 126.	To the extent <i>Lewis</i> does not explicitly disclose this limitation, one of ordinary skill would combine <i>Lewis</i> with <i>Dyer</i> and <i>Proebsting</i> to arrive at the claimed feature. Specifically, <i>Dyer</i> and <i>Proebsting</i> together show that the play execution of byte codes against a memory to identify underlying code, such as the static initialization of an array, was commonly known to those of ordinary skill in the art at the time of the invention, and that it was used to identify the effect of an operation memory. For example, the decompiler technology disclosed by <i>Proebsting</i> discloses a technique for decompilation of Java byte code that play executes the Java virtual machine's stack:	"This paper presents our technique for automatically decompiling Java bytecode into Java source." <i>Proebsting</i> at Abstract.	<i>"Decompilation</i> transforms a low-level language into a high-level language. The Java Virtual machine (JVM) specifies a low-level bytecode language for a stack-based machine. This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a <i>class file</i> that contains type
U.S. Patent No. 6,061,520 – Claim 11 11. The method of claim 6 wherein the operation has an effect on memory, and wherein the play executing step includes the step of: play executing the code to identify the effect on the memory.				

U.S. Patent No. 6,061,520 - Claim 11	Lewis in view of Dyer and further in view of Proebsting
	information and bytecode. The JVM requires a significant amount of type information from the class files for object linking Decompilation systems can exploit this type of information and well-behaved property to recover Java source code from the class file.
	We present a technique for transforming low-level Java bytecode into legal Java source code." <i>Proebsting</i> § 1.
	"We have implemented a prototype Java decompiler, Krakatoa, in Java." Proebsting § 5.
	"Krakatoa uses a stack simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations." <i>Proebsting</i> § 1.
	"Symbolic execution of the bytecode creates the corresponding Java source expressions Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." <i>Proebsting</i> § 2.
	Thus it would have been obvious to one of ordinary skill in the art to combine the methods of <i>Lewis</i> with the play execution (symbolic execution) of bytecode against a memory, as disclosed by <i>Proebsting</i> . This would be for the purpose of "identify[ing] the [code's] effect on memory," as recited in the claim. Further, the artisan would look to known decompilers, such as those disclosed in <i>Dyer</i> . <i>Dyer</i> reviews three popular decompilers that were available before the '520 patent was filed: DejaVu, Mocha, and WingDis. <i>See Dyer</i> at 1. In that review, <i>Dyer</i> analyzes the decompilation of code, including the decompilation of static initializers:
	"Mocha transformed a static initializer into an elegant, but illegal, construction:
	<pre>public ConsoleWindow(String string, int il) { dead = false; styles = { "Plain", "Bold", "Italic" };</pre>

U.S. Patent No. 6,061,520 - Claim 11 Lewis in view of Dyer and further in view of Precising sizes = { """, """, "10", "10", "11", "15", "13", "24" } sizes = { """, "9", "10", "12", "14", "16", "13", "24" } sizes = { """, "9", "10", "12", "14", "16", "13", "24" } Brackcted initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but 1 m sure Sum must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler. When decompling this same static initializer, WingDis produced equally beautiful and syntactically correct code." Dyer and further in view of Proebsting L.S. Patent No. 6.061:520 = Claim 12 Lewis in view of Dyer and further in view of Proebsting U.S. Patent No. 6.061:520 = Claim 12 Lewis discloses a data processing system. cg., the C++ computer programming language, comprising: Lewis discloses a data processing system. cg., the C++ computer programming language, comprising: The Clarity C++ programming language is a dialect of C++ being developed in Sun Microsystems. Pare discloses a data processing. The Clarity C++ programming language is a dialect of C++ being developed in Sun Microsystems. The Clarity C++ programming language is a dialect of C++ being developed in Sun Microsystems. The Clarity restemation of Clarity, we have develo
--

U.S. Patent No. 6,061,520 - Claim 12	Lewis in view of Dyer and further in view of Proebsting
	of a portable, on-the-fly compiler for a subset of the C language; we will refer to this system as 'CCore.'" <i>Lewis</i> at 120.
a storage device containing:	<i>Lewis</i> discloses a storage device. The disclosure of <i>Lewis</i> pertains to a computer system with memory stacks, which are inherently storage devices.
	"Runtime generation of machine code offers many advantages. A runtime code generator can take advantage of information about the particular target platform to generate better code. For example, different implementations of the SPARC architecture have different instruction pipeline properties. In the case of one new SPARC implementation, code generated specifically for this new processor can run up to 25% faster than code generated for a 'generic' SPARC processor. A runtime code generator can also take advantage of the specific values used in a program to generate machine code customized for those values. One of our goals is to explore the use of on-the-fly code generation for systems programming within Sun.
	Our representation of MCode object files is unusual in that we use platform-standard object files instead of some Clarity- or MCode-specific representation. This enables us to fully interoperate with existing C and SPARC ABI code. Object files containing MCode (which we call <i>Linkable MCode</i> files) are processed by standard linkers and other tools in the same way as other object files. As an example, our Solaris SPARC implementation supports complete interoperation with all SPARC Application Binary Interface (ABI) compliant code [SPARC ABI]. In particular, interpreted or compiled MCode programs can call C programs and vice versa, addresses of MCode procedures can be passed to C code and later called, and all C data types can be exchanged." <i>Lewis</i> at 119.
a program with source code that statically initializes a data structure; and	<i>Lewis</i> discloses a program with source code that statically initializes a data structure. For example, <i>Lewis</i> discloses MCode which may contain instruction to initialize a data structure.
	"Although MCode includes instructions and data structures needed to implement some Clarity language-specific constructs such as its exceptions and method calls, the core of MCode is suitable for representing code for C and many other languages." <i>Lewis</i> at 119.

	_	
	.H	
	4	
	tn	
	2	
	a	
	st	
	<u> </u>	
	g	
00	2	
2	문	
1	G	
\$	l g l	
P.	H L	
5	► <u>></u>	
` ?	l ∪ ∷. ∣	
.	5 A L	
	<u></u>	
5		
•	<u> </u>	
2	9 HE I	
5	- <u>5</u>	
	Lu	
	·	
-	. tl	
	ы. Б. С. – Г	
<u> </u>	မြိုင်းမြို	
- -	∣ ਠੱ;ਹੀ	
Ē	⊢ o.¥I	
'E	2 원	
, 7	<u> </u>	
-	ା ନୁହା	
70	। ਸ਼ੁਸ਼ਾ	
2	s a	
Lewis in view of Dyer and further in view of Proebsting	ା ଛ ଛା	
<u> </u>	a XI	
0	6.9	
	t 🗠	
Q	L F	
	이드	
8	to to	
~		
5		
O	୍ ଅ ତ	
•5	qa	
	X O	
	ၿပ	
•	Ξ Ū	
	A 8	
Ş	00	
5	Ĕ Ă	
Ň	- E -	
	ယက္ရ	
	<u>6</u>	
	2 4	
	ਿ ਸ਼ ਨੇ ਕਿ	
	b v l	
	ŝõ	
	8 21	
	Figures 5 and 6 give an example of generated MCode. The Clarity method <i>startup</i> in igure 5 produces the MCode instructions shown in part in Figure 6."	
	ଇ ମ	
	<u>ස</u> හ	
	F. F.	
	, <u>1</u>	
2		
2		
5		
-		
2		
0,		
Ó		
•		
•		
7		
l 🖬 🕬		
1		
4		
1 H H		
à		
å		
S. P:		
.S. P.		
U.S. Patent No. 6,061,520 - Claim 12		

U.S. Patent No. 6,061.520 – Claim 12 Zewis in view of Dyer and further in view of <i>Provisiting</i> work, mute: "Freads interface the set of the s

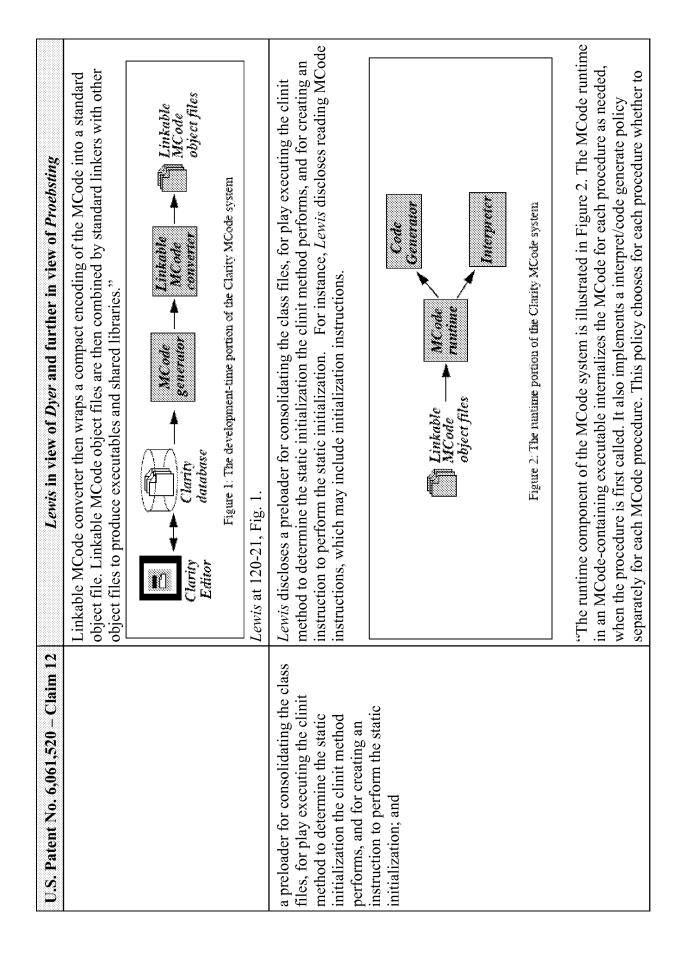
Lewis in view of Dyer and further in view of Proebsting	<i>Lewis</i> discloses class files, wherein one of the class files contains a clinit method that statically initializes the data structure. This subsection presents an overview of the Clarity MCode compilation system; more details are given in Section 3. There are two major parts to the MCode compilation system: a program development-time part and a runtime part. The development-time portion consists of an MCode generator for the Clarity language and a Linkable MCode converter; see Figure 1. The MCode generator reads semantically decorated Clarity ASTs stored in the Clarity programming environment's database and produces platform-independent MCode. The Linkable MCode converter then wraps a compact encoding of the MCode into a standard object file. Linkable MCode object files are then combined by standard linkers with other object files to produce executables and shared libraries."	Editor Figure 1: The development-time portion of the Clarity MCode system	<i>Lewis</i> at 120-21, Fig. 1.	"Linkable MCode object files contain a machine-independent <i>pickle</i> of an MCode code unit. This is a compact, platform-independent encoding of the MCode information into a sequence of bytes. This pickle can later be internalized or unpickled to reconstruct the original MCode. The MCode for each procedure is pickled separately to support procedure-at-a-time processing. The current encoding is not especially compact although the Linkable MCode object files are still smaller than object files containing machine code. We intend to replace the current pickle format with a more compact one.	Linkable MCode object files are platform-standard object files that are processed in the usual way by the platform's standard linker. This means they need to include platform-dependent definitions of global variables and procedures, and descriptions of referenced symbols. We
U.S. Patent No. 6,061,520 - Claim 12	class files, wherein one of the class files contains a clinit method that statically initializes the data structure; initializes the data structure (

Lewis in view of Dyer and further in view of Proebsting	currently encode ('mangle') symbol names in order to ensure that the resulting executables or shared libraries are type-safe with respect to the Clarity language. Eventually, this type-safety will be checked by a Clarity <i>prelinker</i> .	Besides symbol definitions and references, our Solaris Linkable MCode object files also contain a few machine language instructions for each procedure's entry code. This entry code allows C code to call the MCode procedure. On the SPARC, this entry code consists primarily of a three instruction 'trampoline' that redirects the call to the appropriate target procedure chosen by the interpret/compile strategy module in the MCode runtime. The SPARC entry code also has three words used when atomically updating the trampoline. Despite this platform-specific information, the contents of a Linkable MCode file are mostly platform-independent. The Linkable MCode converter itself is also mostly platform-independent. We currently execute the Linkable MCode converter during program development, before a program is distributed. It could also be executed when the program is installed on a particular platform." <i>Lewis</i> at 125-26.
U.S. Patent No. 6,061,520 - Claim 12		

Lewis in view of Dyer and further in view of Proebsting	startup: method (our_workers: in int) { // executed when the thread is started; delegates most of its work to forked sitking worklers within work_mutex { // acquire work_mutex for duration of the within statement	LosdGlotat(0) work_mutex	invokeOuter(0x02030204) Threadtenter	our_work = work_per_worker; *	Laadelosaa(1) work_per_worker StoreGiobiai(5) our_work	LoadGlohai(3) extra_work	Losaint(0, 0) integer constant 0, type 0	Compareint(>) extra_work>0	20nd_fa\se_tag_2)	Curr_wrotk += 1;	GiobalAddr(5) our_work	Dup()	Loadindirect(D)	D, 4)	Addint() our_work+=1	Storefindisect(D) our_work	GiobalAddr(3) extra_work	Bup() Loadindiect(0)	LoadSigned(0, 1) integer constant 1, type 0	Storeindirect(0) extra_work	BeginEise(cond_faise_tag_2) Enditored if the 20	άτι,
U.S. Patent No. 6,061,520 - Claim 12																						

	•	
	LoadGlohai(2) invokeCuter(0x02010205)	work_mutex Threadtexit
	defegated_to_workers: int = (our_workers - 1); LoadArg(0)	
	Losant(J. 1) Susint() 	Integer constant 1, type 2 our_workers-1
	Store Localu?	cered_ro_workers
	eo_work(our_work); LeoadGlobai(\$}	do work
	LoadGlobai(5) ProcCail(7)	our_work do_work(our_work)
	እ ProcReturn(፤)	method(our_workers
	Figure 6: MCode instructions generated	Figure 6: MCode instructions generated for the $\mu 0^{++}$ test program's startup method
	<i>Lewis</i> at 125, Fig. 6.	
To	the extent <i>Lewis</i> does not explicitly discill in the art would combine the disclosun ature. Specifically, <i>Dyer</i> discloses deco	To the extent <i>Lewis</i> does not explicitly disclose this limitation of Claim 12, one of ordinary skill in the art would combine the disclosure of <i>Lewis</i> with <i>Dyer</i> to arrive at the claimed feature. Specifically, <i>Dyer</i> discloses decompiling static array initialization statements.
Se ini	<i>See Dyer</i> at 3. The '520 patent is clear that initialization bytecode into the clinit method:	See Dyer at 3. The `520 patent is clear that the prior art Java compiler puts all static array initialization bytecode into the clinit method:
, A.'	"As stated above, the class file format can initialize arrays. To compensate for this j method, <clinit>, to perform class initializa .520 patent at 1.57_61</clinit>	"As stated above, the class file format cannot instruct the virtual machine to statically initialize arrays. To compensate for this problem, the Java compiler generates a special method, <clinit>, to perform class initialization, including initialization of static arrays."</clinit>
Ac	Accordingly, for the decompilation of static array initialization str <i>Dyer</i> , the bytecode is necessarily in the clinit method, as claimed.	Accordingly, for the decompilation of static array initialization statements, as disclosed in <i>Dyer</i> , the bytecode is necessarily in the clinit method, as claimed.
a memory containing:	Lewis discloses a memory.	
'R tak	untime generation of machine code offe ce advantage of information about the pa	"Runtime generation of machine code offers many advantages. A runtime code generator can take advantage of information about the particular target platform to generate better code.

U.S. Patent No. 6,061,520 - Claim 12	Lewis in view of Dyer and further in view of Proebsting
	For example, different implementations of the SPARC architecture have different instruction pipeline properties. In the case of one new SPARC implementation, code generated specifically for this new processor can run up to 25% faster than code generated for a 'generic' SPARC processor. A runtime code generator can also take advantage of the specific values used in a program to generate machine code customized for those values. One of our goals is to explore the use of on-the-fly code generation for systems programming within Sun.
	Our representation of MCode object files is unusual in that we use platform-standard object files instead of some Clarity- or MCode-specific representation. This enables us to fully interoperate with existing C and SPARC ABI code. Object files containing MCode (which we call <i>Linkable MCode</i> files) are processed by standard linkers and other tools in the same way as other object files. As an example, our Solaris SPARC implementation supports complete interoperation with all SPARC Application Binary Interface (ABI) compliant code [SPARC ABI]. In particular, interpreted or compiled MCode programs can call C programs and vice versa, addresses of MCode procedures can be passed to C code and later called, and all C data types can be exchanged." <i>Lewis</i> at 119.
	To the extent <i>Lewis</i> does not explicitly disclose this limitation of Claim 12, one of ordinary skill in the art would combine the disclosure of <i>Lewis</i> with <i>Dyer</i> and <i>Proebsting</i> to arrive at the claimed feature. Specifically, all of these disclosed software systems necessarily disclose a memory.
a compiler for compiling the program and generating the class files; and	<i>Lewis</i> discloses a compiler as a part of the "MCode compilation system" for compiling the program and generating the class files.
	"This subsection presents an overview of the Clarity MCode compilation system; more details are given in Section 3. There are two major parts to the MCode compilation system: a program development-time part and a runtime part. The development-time portion consists of an MCode generator for the Clarity language and a Linkable MCode converter; see Figure 1. The MCode generator reads semantically decorated Clarity ASTs stored in the Clarity programming environment's database and produces platform-independent MCode. The



U.S. Patent No. 6,061,520 - Claim 12	Lewis in view of Dyer and further in view of Proebsting
	This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a <i>class file</i> that contains type information and bytecode. The JVM requires a significant amount of type information from the class files for object linking Decompilation systems can exploit this type of information and well-behaved property to recover Java source code from the class file.
	We present a technique for transforming low-level Java bytecode into legal Java source code." <i>Proebsting</i> § 1.
	"We have implemented a prototype Java decompiler, Krakatoa, in Java." Proebsting § 5.
	"Krakatoa uses a stack simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations." <i>Proebsting</i> § 1.
	"Symbolic execution of the bytecode creates the corresponding Java source expressions Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." <i>Proebsting</i> § 2.
	Thus it would have been obvious to one of ordinary skill in the art to combine the methods of <i>Lewis</i> with the play execution (symbolic execution) of bytecode against a memory, as disclosed by <i>Proebsting</i> . And, even more specifically, the artisan would look to known decompilers, such as those disclosed in <i>Dyer</i> . <i>Dyer</i> reviews three popular decompilers that were available before the '520 patent was filed: DejaVu, Mocha, and WingDis. <i>See Dyer</i> at 1. In that review, <i>Dyer</i> analyzes the decompilation of code, including the decompilation of static initializers:
	"Mocha transformed a static initializer into an elegant, but illegal, construction:
	<pre>public ConsoleWindow(String string, int il) { dead = false;</pre>

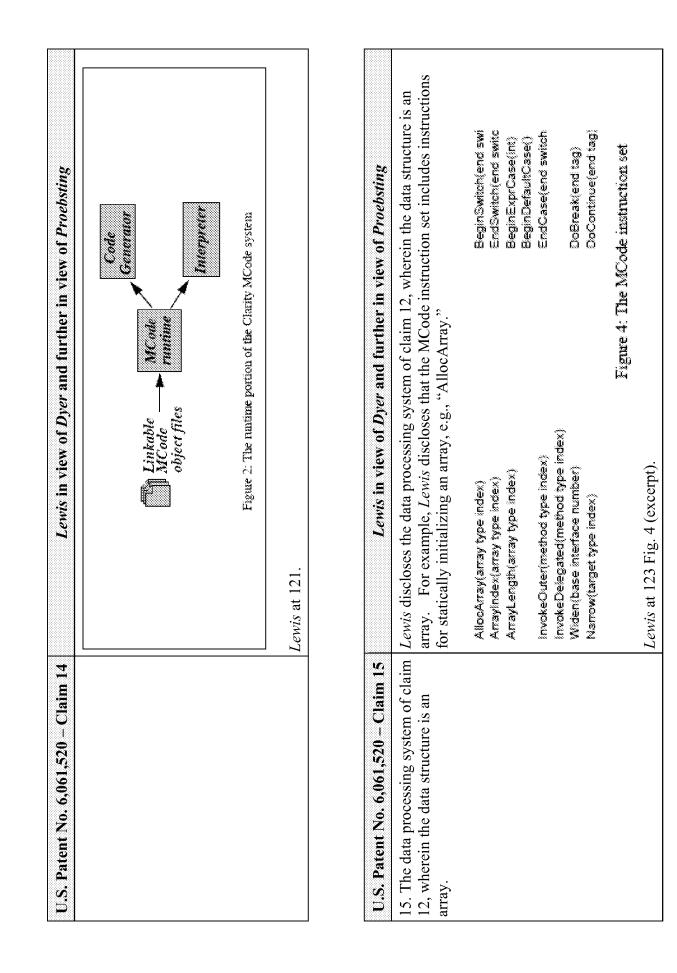
U.S. Patent No. 6,061,520 - Claim 12	Lewis in view of Dyer and further in view of Proebsting
	styles = { "Plain", "Bold", "Italic" }; sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." <i>Dyer</i> at 3.
	Further, <i>Proebsting</i> discloses that the Krakatoa decompiler writes the source code (which would necessarily include creating any static array initialization instructions) to an output file:
	"We have implemented a prototype Java decompiler, Krakatoa, in Java. We have run Krakatoa on a number of class file, including some to which we had no source code access. We examined the output of Krakatoa by hand, and Krakatoa appears to recover high-level constructs very well." <i>Proebsting</i> § 5.
	"Krakatoa is very efficient at reproducing readable Java source from Java bytecode." Proebsting § 6.
	Additionally, <i>Dyer</i> discloses that the Java decompiler (such as <i>Proebsting's</i> Krakatoa decompiler) creates an instruction that "perform[s] the static initialization," as recited in the claim. <i>See Dyer</i> at 3.
	The fact that some of $Dyer$'s decompiled code may be considered "illegal" should not detract from the fact that the decompilation of static array initialization byte code into a single expression or instruction was a well-known technique at the time the '520 patent was filed

2 Lewis in view of Dyer and further in view of Proebsting	a technique used to identify the operation of the byte code in order to generate equivalent high-level language expressions. In fact, the '520 patent discloses creating a constant pool entry that is not a standard Java virtual machine construct. Thus, one of ordinary skill in the art would have considered this constant pool construct (i.e., CONSTANT_Array (<i>see</i> '520 patent at 8:54-9:13)) as an "illegal" Java constant pool entry. The reason it wouldn't have mattered is that the '520 patent states that "the virtual machine 222 is modified to recognize the static initialization directives of the preloader." '520 patent at 4:46-48. Because the exact form of modification is not disclosed, it must be within the ordinary artisan's skill set to perform the correct modification to allow the virtual machine to recognize an "illegal" instruction such as the new CONSTANT_Array type.	Further, during Examination claim terms are to be given their broadest reasonable interpretation (<i>see</i> MPEP § 2111), and there are no qualifiers in the claim language as to the generated instruction. Again, because the '520 patent assumes that one of ordinary skill in the art would modify the virtual machine to recognize the generated instruction, the exact format (whether syntactically correct or not) is not an issue. And in any case, <i>Dyer</i> discloses that in contrast to the Mocha decompiler's "illegal" code the WingDis decompiler produced "syntactically correct code." <i>Dyer</i> at 3.	<i>Lewis</i> discloses a processor for running the compiler and the preloader. <i>Lewis</i> discloses the implementation of code that can be generated specifically for certain processors.	"Runtime generation of machine code offers many advantages. A runtime code generator can take advantage of information about the particular target platform to generate better code. For example, different implementations of the SPARC architecture have different instruction pipeline properties. In the case of one new SPARC implementation, code generated specifically for this new processor can run up to 25% faster than code generated for a 'generic' SPARC processor. A runtime code generator can also take advantage of the specific values used in a program to generate machine code customized for those values. One of our goals is to explore the use of on-the-fly code generation for systems programming within Sun.
U.S. Patent No. 6,061,520 - Claim 12			a processor for running the compiler and the preloader.	

U.S. Patent No. 6,061,520 - Claim 12	Lewis in view of Dyer and further in view of Proebsting
	Our representation of MCode object files is unusual in that we use platform-standard object files instead of some Clarity- or MCode-specific representation. This enables us to fully interoperate with existing C and SPARC ABI code. Object files containing MCode (which we call <i>Linkable MCode</i> files) are processed by standard linkers and other tools in the same way as other object files. As an example, our Solaris SPARC implementation supports complete interoperation with all SPARC Application Binary Interface (ABI) compliant code [SPARC ABI]. In particular, interpreted or compiled MCode programs can call C programs and vice versa, addresses of MCode procedures can be passed to C code and later called, and all C data types can be exchanged." <i>Lewis</i> at 119.
	To the extent <i>Lewis</i> does not explicitly disclose this limitation of Claim 12, one of ordinary skill in the art would combine the disclosure of <i>Lewis</i> with <i>Dyer</i> and <i>Proebsting</i> to arrive at the claimed feature. Specifically, all of these disclosed software systems necessarily require a processor to run the disclosed software.
U.S. Patent No. 6,061,520 - Claim 13	Lewis in view of Dyer and further in view of Proebsting
claim a out file	<i>Lewis</i> discloses the data processing system of claim 12 wherein the preloader includes a mechanism for generating an output file containing the created instruction, i.e., a code generator that generates good code for SPARC.
	"The code generator includes a peephole optimizer, does dead code elimination, and generates "leaf procedure" calls on the SPARC. However, little further optimization is done at this time; our immediate concern is generating correct code. Despite this, the code generator generator generates good code for the SPARC." <i>Lewis</i> at 126.
	To the extent <i>Lewis</i> does not explicitly disclose this limitation, one of ordinary skill would combine <i>Lewis</i> with <i>Proebsting</i> and <i>Dyer</i> to arrive at the claimed feature. Specifically, <i>Proebsting</i> discloses that the Krakatoa decompiler writes the source code (which would necessarily include any static array initialization instructions) to an output file:

U.S. Patent No. 6,061,520 - Claim 13	Lewis in view of Dyer and further in view of Proebsting
	"We have implemented a prototype Java decompiler, Krakatoa, in Java. We have run Krakatoa on a number of class file, including some to which we had no source code access. We examined the output of Krakatoa by hand, and Krakatoa appears to recover high-level constructs very well." <i>Proebsting</i> § 5.
	"Krakatoa is very efficient at reproducing readable Java source from Java bytecode." Proebsting § 6.
	<i>Dyer</i> discloses that the output of a decompiler (such as <i>Proebsting's</i> Krakatoa decompiler) may be an instruction that "request[s] the static initialization of the array," as recited in the claim:
	"Mocha transformed a static initializer into an elegant, but illegal, construction:
	<pre>public ConsoleWindow(String string, int il) {</pre>
	<pre>dead = false; styles = { "Plain", "Bold", "Italic" }; sizes = { "8", "9", "10", "12", "14", "16", "18", "24" }; </pre>
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." <i>Dyer</i> at 3.
	The fact that some of $Dyer$'s decompiled code may be considered "illegal" should not detract from the fact that the decompilation of static array initialization byte code into a single expression or instruction was a well-known technique at the time the '520 patent was

U.S. Patent No. 6,061,520 - Claim 14	Lewis in view of Dyer and further in view of Proebsting
	procedure's machine language entry code." <i>Lewis</i> at 127.
	"While the MCode interpreter is mostly platform-independent, about 20% of its code is platform-specific. For example, in order to fully support procedure interposition and other
	ABI capabilities, the SPARC MCode interpreter does not directly interpret MCode ProcCall or Invoke instructions but instead implements them as SPARC ABI calls. Even MCode calls
	to other MCode procedures are implemented using SPARC instructions and execute the procedure's machine language entry code. This is necessary because the interpreter can
	never know whether a called procedure is actually implemented in MCode or in C. (For example, a programmer might have replaced the called procedure using interposition.) This
	routine will return an aggregate value, the interpreter must generate a sequence of machine
	instructions at runtime (a <i>thunk</i>) to support the SPARC ABI's calling convention that the returned aggregate's length must be encoded into a SPARC UNIMP instruction just after the call. The intermreter also stores all prooram values in memory as SPARC values since this is
	required for ABI interoperation.
	Recently, a second MCode interpreter has been developed by Mick Jordan. This interpreter executes <i>system models</i> written in the Clarity language. These system models precisely
	describe how a software system is built: the exact versions of its component parts, all options and build parameters, and how the component parts are assembled. This system modeller is
	intended to replace the Unix <i>make</i> tool and to eliminate some of its problems: e.g. the inability to exactly reproduce the construction of a software system. The system modeller's
	MCode interpreter is specialized to executing these models and to interacting with the
	Clarity program database. It does not need, for example, to support SPARC ABI interomeration." <i>Lewis</i> at 127-28.



U.S. Patent No. 6,061,520 - Claim 15	Lewis in view of Dyer and further in view of Proebsting
	To the extent <i>Lewis</i> does not explicitly disclose this limitation, one of ordinary skill would combine <i>Lewis</i> with <i>Dyer</i> to arrive at the claimed feature. Specifically, <i>Dyer</i> discloses that the statically initialized data structure is an array:
	"Mocha transformed a static initializer into an elegant, but illegal, construction:
	<pre>public ConsoleWindow(String string, int il) {</pre>
	<pre>dead = false; styles = { "Plain", "Bold", "Italic" }; sizes = { "8", "9", "10", "12", "14", "16", "18", "24" }; </pre>
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." $Dyer$ at 3.
U.S. Patent No. 6,061,520 – Claim 16	Lewis in view of Dyer and further in view of Proebsting
16. The data processing system of claim 12 wherein the clinit method has byte codes that statically initialize the data structure.	<i>Lewis</i> discloses the data processing system of claim 12 wherein the clinit method has byte codes that statically initialize the data structure. For example, <i>Lewis</i> discloses that the MCode instruction set includes instructions for statically initializing an array, e.g., "AllocArray."

U.S. Patent No. 6,061,520 Claim 16 Lowis in view of Dyer and (urther in view of Prochsting AlcoArray/array type index) BeginSentotiend swit Array head (array type index) BeginSentotiend swit Array index/array type index) BeginSentotiend swit Array index/array type index) BeginSentotiend swit Answeb-beginsentor type index) BeginSentotiend swit Mene(Dase interface number) Defrastformethod systemethod type index) National array: Defrastformation set Lewis ar 123 Fig. 4 (excerpt) Defrastform set Lewis ar 123 Fig. 4 (excerpt) Defrastform set None for the extent Lewis does not explicitly disclose this limitation, one of ordinary skill would combine Lewis with Dyper to arrive at the claimed feature. Specifically, Dyer discloses that the statically initializer into an elegant, but illegal, construction: Dublic Consolewindow (String string, int ill) Moch attansformed a static initializer into an elegant, but illegal, construction: public Consolewindow (String string, int ill) Moch attansformed a structure is an array: motion is a finitializer lists for arrays are valid only as initializers for variable declarations (string string, int ill) fead = false) String, int ill, ill, ill, ill, ill, ill, ill, ill

U.S. Patent No. 6,061,520 - Claim 16	Lewis in view of Dyer and further in view of Proebsting
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." $Dyer$ at 3.
	Additionally, the '520 patent is clear that the prior art Java compiler puts all static array initialization bytecode into the clinit method:
	"As stated above, the class file format cannot instruct the virtual machine to statically initialize arrays. To compensate for this problem, the Java compiler generates a special method, <clinit>, to perform class initialization, including initialization of static arrays." '520 patent at 1:57-61.</clinit>
	Accordingly, for the decompilation of static array initialization statements, as disclosed in <i>Dyer</i> , the bytecode is necessarily in the clinit method, as claimed.
U.S. Patent No. 6,061,520 - Claim 17	Lewis in view of Dyer and further in view of Proebsting
17. The data processing system of claim 12 wherein the created instruction includes an entry into a constant pool.	<i>Lewis</i> discloses the data processing system of claim 12, wherein the created instruction includes an entry into a constant pool. For example, <i>Lewis</i> discloses the entry of a constant as a "CGValue." "The second C++ base class, CGValue, describes values during compilation. The code generator "executes" MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously "executed" subexpressions, and procedure or method calls." <i>Lewis</i> at 126.
	standard Java virtual machine," <i>see</i> '520 patent at 7:48-49, and wherein the "virtual machines recognize various constant pool entries, such as CONSTANT_Integer, CONSTANT_String, and CONSTANT Long," <i>see</i> '520 patent at 7:51-54. The purportedly inventive step involves inserting a "CONSTANT_Array entry in the constant pool." '520

U.S. Patent No. 6,061,520 - Claim 17	Lewis in view of Dyer and further in view of Proebsting
	Linkable MCode object files are still smaller than object files containing machine code. We intend to replace the current pickle format with a more compact one.
	Linkable MCode object files are platform-standard object files that are processed in the
	platform-dependent definitions of global variables and procedures, and descriptions of referenced symbols. We currently encode ('mangle') symbol names in order to ensure that
	the resulting executables or shared libraries are type-safe with respect to the Clarity
	language. Eventually, this type-safety will be checked by a Clarity <i>prelinker</i> . Besides symbol definitions and references, our Solaris Linkable MCode object files also contain a
	few machine language instructions for each procedure's entry code. This entry code allows C code to call the MCode procedure. On the SPARC, this entry code consists primarily of a
	three instruction 'trampoline' that redirects the call to the appropriate target procedure
	cnosen by the interpret/compile strategy module in the MC one runtime. The SFARC entry code also has three words used when atomically updating the trampoline. Despite this
	platform-specific information, the contents of a Linkable MCode file are mostly platform-independent. The Linkable MCode converter itself is also mostly
	platform-independent. We currently execute the Linkable MCode converter during program
	development, before a program is distributed. It could also be executed when the program is installed on a particular platform." <i>Lewis</i> at 125-26.
U.S. Patent No. 6,061,520 - Claim 18	Lewis in view of Dyer and further in view of Proebsting
18. A computer-readable medium	<i>Lewis</i> discloses a computer-readable medium containing instructions for controlling a data processing system to perform a method, e.g., the C++ language and certain dialects thereof.
data processing system to perform a method, comprising the steps of:	capable of performing data processing and performing a method.
)	"The <i>Clarity</i> C++ programming language is a dialect of C++ being developed in Sun Microsystems I aboratories Clarity shares many features with C++ but is less complex and
	has a more consistent syntax and simpler semantics without loss in expressiveness Clarity is intended to be a wide-spectrum language suitable for both systems and application

U.S. Patent No. 6,061,520 – Claim 18	<i>Lewis</i> in view of <i>Dyer</i> and further in view of <i>Proebsting</i>
prog	programming, particularly of distributed software.
To st	To support the compilation of Clarity, we have developed a high-level, machine-independent
inter-	intermediate representation that we call <i>MCode</i> (for "middle code"). We use MCode to
inter-	compile Clarity programs at execution time (i.e., on-the-fly) into SPARC code for the
comp	Solaris operating system. This code generator is designed to be largely machine
Solat	independent: besides the SPARC code generator, an Intel x86 version is being developed."
indep	<i>Lewis</i> at 119 (footnote omitted).
<i>Lewi</i>	"MCode has its basis in unpublished work done by L. Peter Deutsch at Sun Microsystems
as 'C	Laboratories in 1992-93. This work consisted of an implementation in Smalltalk of the core
receiving code to be run on a processing <i>Lewi</i>	of a portable, on-the-fly compiler for a subset of the C language; we will refer to this system
component to perform an operation;	as 'CCore.'" <i>Lewis</i> at 120.
Sec t	<i>Lewis</i> at 120.
	Code system

Lewis in view of Dyer and further in view of Proebsting	Linkable MC ode MC ode munime object files	Figure 2: The mutime portion of the Clarity MCode system	"This subsection presents an overview of the Clarity MCode compilation system; more details are given in Section 3. There are two major parts to the MCode compilation system: a program development-time part and a runtime part. The development-time portion consists of an MCode generator for the Clarity language and a Linkable MCode converter; see Figure 1. The MCode generator reads semantically decorated Clarity ASTs stored in the Clarity programming environment's database and produces platform-independent MCode. The Linkable MCode converter then wraps a compact encoding of the MCode into a standard object files to produce executables and shared libraries.	The runtime component of the MCode system is illustrated in Figure 2. The MCode runtime in an MCode-containing executable internalizes the MCode for each procedure as needed, when the procedure is first called. It also implements a interpret/code generate policy separately for each MCode procedure. This policy chooses for each procedure whether to interpret it, generate code, or interpret then later generate code, or generate better code. The code generator currently produces SPARC code of approximately the quality of the SunPRO C compiler at the default -O2 optimization level. A port of the code generator to the x86 is underway. The MCode interpreter interoperates with all SPARC ABI code. Like the compiler, it is reentrant and supports multithreaded programs. It also does extensive checking during program execution, which makes it especially useful for uncovering errors
U.S. Patent No. 6,061,520 - Claim 18				

U.S. Patent No. 6,061,520 - Claim 18	Lewis in view of Dyer and further in view of Proebsting
	in Clarity programs that are otherwise difficult to detect. The interpreter will also be used by the Clarity debugger that we are developing to evaluate Clarity statements and expressions." <i>Lewis</i> at 120-21.
simulating execution of the code without running the code on the processing component to identify the operation if the code were run by the	<i>Lewis</i> discloses simulating execution of the code without running the code on the processing component to identify the operation if the code were run by the processing component. See the excerpts below and the discussion above as to claim 1.
processing component; and	"The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGV alue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously "executed" subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." <i>Lewis</i> at 126.
	To the extent <i>Lewis</i> does not explicitly disclose this limitation, one of ordinary skill would combine <i>Lewis</i> with <i>Dyer</i> and <i>Proebsting</i> to arrive at the claimed feature. Specifically, <i>Dyer</i> and <i>Proebsting</i> together show that the simulation of byte codes against a memory to identify underlying code, such as the static initialization of an array, was commonly known to those of ordinary skill in the art at the time of the invention. For example, the decompiler technology disclosed by <i>Proebsting</i> discloses a technique for decompilation of Java byte code that simulates the Java virtual machine's stack:
	"This paper presents our technique for automatically decompiling Java bytecode into Java source." <i>Proebsting</i> at Abstract.
	" <i>Decompilation</i> transforms a low-level language into a high-level language. The Java Virtual machine (JVM) specifies a low-level bytecode language for a stack-based machine. This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a <i>class file</i> that contains type information and bytecode. The JVM requires a significant amount of type information

U.S. Patent No. 6,061,520 - Claim 18	Lewis in view of Dyer and further in view of Proebsting
	from the class files for object linking Decompilation systems can exploit this type of information and well-behaved property to recover Java source code from the class file.
	We present a technique for transforming low-level Java bytecode into legal Java source code." <i>Proebsting</i> § 1.
	"We have implemented a prototype Java decompiler, Krakatoa, in Java." Proebsting § 5.
	"Krakatoa uses a stack simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations." <i>Proebsting</i> § 1.
	"Symbolic execution of the bytecode creates the corresponding Java source expressions Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." <i>Proebsting</i> § 2.
	Thus it would have been obvious to one of ordinary skill in the art to combine the methods of <i>Lewis</i> with the simulation (symbolic execution) of bytecode against a memory, as disclosed by <i>Proebsting</i> . And, even more specifically, the artisan would look to known decompilers, such as those disclosed in <i>Dyer</i> . <i>Dyer</i> reviews three popular decompilers that were available before the '520 patent was filed: DejaVu, Mocha, and WingDis. <i>See Dyer</i> at 1. In that review, <i>Dyer</i> analyzes the decompilation of code, including the decompilation of static initializers:
	"Mocha transformed a static initializer into an elegant, but illegal, construction:
	<pre>c ConsoleWindow(String string, int i1) = false; s = { "Plain", "Bold", "Italic" }; </pre>
	sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };

U.S. Patent No. 6,061,520 - Claim 18	Lewis in view of Dyer and further in view of Proebsting
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." <i>Dyer</i> at 3.
	The fact that some of <i>Dyer's</i> decompiled code may be considered "illegal" should not detract from the fact that the decompilation of static array initialization byte code into a single expression or instruction was a well-known technique at the time the '520 patent was filed – a technique used to identify the operation of the byte code in order to generate equivalent high-level language expressions. In fact, the '520 patent discloses creating a constant pool entry that is not a standard Java virtual machine construct. Thus, one of ordinary skill in the art would have considered this constant pool entry. The vary (<i>see '520</i> patent at 8:54-9:13)) as an "illegal" Java constant pool entry. The reason it wouldn't have mattered is that the '520 patent states that "the virtual machine 222 is modified to recognize the static initialization directives of the preloader." '520 patent at 4:46-48. Because the exact form of modification is not disclosed, it must be within the ordinary artisan's skill set to perform the correct modification to allow the virtual machine to recognize an "illegal" instruction such as the new CONSTANT_Array type. Further, during Examination claim terms are to be given their broadest reasonable interpretation (<i>see</i> MPEP § 2111), and there are no qualifiers in the claim language as to the generated instruction. Again, because the '520 patent assumes that one of ordinary skill in the art would modify the virtual machine to recognize the generated instruction, the exact form at (whether syntactically correct or not) is not an issue. And in any case, <i>Dyer</i> discloses that - in contrast to the Mocha decompiler's "Illegal" code - the WingDis decompiler produced "syntactically correct code." <i>Dyer</i> at 3.
creating an instruction for the	<i>Lewis</i> discloses creating an instruction for the processing component to perform the

U.S. Patent No. 6,061,520 - Claim 18 processing component to perform the operation.	<i>Lewis</i> in view of <i>Dyer</i> and further in view of <i>Proebsting</i> operation. See the excerpts below and the discussion above as to claim 1. "The object-oriented architecture of the code generator is designed to be retargetable to a new implementation. The MCode machine code generator is designed to be retargetable to a new machine architecture (especially a RISC machine) with relatively little effort. It defines two key C++ base classes that must be subclassed to port the code generator. The first class, CGiMachine, represents a target machine for code generaton and a code stream for that machine. The basic machine model is a generic, nonwindowed RISC processor. CGMachine implement virtual methods that describe the target machine's registers, data types, and instruction properties. CGMachine methods then use those descriptions to generate machine code from MCode The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Chorerets subclasses include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that walte of those expressions is needed." <i>Lewis</i> at 126. "In order to do high-quality code generation, we need to rebuild the expression trees from the stack machine. The code generation, we need to rebuild the final target for an expression is known. Much of the process is similar to that of TNBIND algorithm documented in [Wulf 75]. This algorithm gives excellent results and excentes extremely efficiently. A final optimization does instruction reordering to minimize RISC processor
	province execution contricts. By organizing the code generator into a series of caseading object streams, we are able to consume MCode and generate native machine code in one pass. Our object-oriented architecture provides an efficient way to trade increased memory for speed." $Lewis$ at 127.

U.S. Patent No. 6,061,520 - Claim 18	Lewis in view of Dyer and further in view of Proebsting
	"We have described an intermediate representation MCode that is compact, easy to generate, and supports the on-the-fly generation of good quality machine code. Linkable MCode is an encoding of MCode in platform-standard object files that enables full interoperation with C and existing libraries, as well as the full use of all capabilities of standard linkers and other programming tools." <i>Lewis</i> at 128.
	To the extent <i>Lewis</i> does not explicitly disclose this limitation, one of ordinary skill would combine <i>Lewis</i> with <i>Proebsting</i> and <i>Dyer</i> to arrive at the claimed feature. Specifically, <i>Proebsting</i> discloses that the Krakatoa decompiler writes the source code (which would necessarily include creating any static array initialization instructions) to an output file:
	"We have implemented a prototype Java decompiler, Krakatoa, in Java. We have run Krakatoa on a number of class file, including some to which we had no source code access. We examined the output of Krakatoa by hand, and Krakatoa appears to recover high-level constructs very well." <i>Proebsting</i> § 5.
	"Krakatoa is very efficient at reproducing readable Java source from Java bytecode." Proebsting § 6.
	<i>Dyer</i> discloses that the output of a decompiler (such as <i>Proebsting</i> 's Krakatoa decompiler) may be an instruction that "request[s] the static initialization of the array," as recited in the claim:
	"Mocha transformed a static initializer into an elegant, but illegal, construction:
	3
	styles = { "Flain", "Bold", "Italic" }; sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations

U.S. Patent No. 6,061,520 - Claim 18	Lewis in view of Dyer and further in view of Proebsting
	(either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." <i>Dyer</i> at 3.
	The fact that some of $Dyer's$ decompiled code may be considered "illegal" should not detract from the fact that the decompilation of static array initialization byte code into a single expression or instruction was a well-known technique at the time the '520 patent was filed a technique used to identify the operation of the byte code in order to generate
	equivalent high-level language expressions. In fact, the '520 patent discloses creating a constant pool entry that is not a standard Java virtual machine construct. Thus, one of ordinary skill in the art would have considered this constant pool construct (i.e., CONSTANT Array (<i>see</i> '520 patent at 8:54-9:13)) as an "illegal" Java constant pool entry.
	The reason it wouldn't have mattered is that the '520 patent states that "the virtual machine 222 is modified to recognize the static initialization directives of the preloader." '520 patent at 4:46-48. Because the exact form of modification is not disclosed, it must be within the ordinary artisan's skill set to perform the correct modification to allow the virtual machine to recognize an "illegal" instruction such as the new CONSTANT_Array type.
	Further, during Examination claim terms are to be given their broadest reasonable interpretation (<i>see</i> MPEP § 2111), and there are no qualifiers in the claim language as to the generated instruction. Again, because the '520 patent assumes that one of ordinary skill in the art would modify the virtual machine to recognize the generated instruction, the exact format (whether syntactically correct or not) is not an issue. And in any case, <i>Dyer</i> discloses that in contrast to the Mocha decompiler's 'illegal'' code the WingDis decompiler produced "syntactically correct code." <i>Dyer</i> at 3.
	Additionally, the recitation of this method step in Claim 1 of the '520 patent is not limited with respect to the order in which it is performed relative to the other steps of Claim 1.

 520 – Claim 18 Lewis in view of Dyer and further in view of Proebsting Thus, the combined systems of Lewis, Proebsting, and Dyer may perform this step (i.e., "creating an instruction") after it performs the step of "receiving code" and before the step of "simulating execution." Or it may perform this step after "simulating execution." Either scenario would read on the claim limitations, which, under a broadest reasonable interpretation (MPEP § 2111), do not require a certain order. 	520 – Claim 19 <i>Lewis</i> in view of <i>Dyer</i> and further in view of <i>Proebsting</i>	ble medium of <i>Lewis</i> discloses the computer-readable medium of claim 18 wherein the operation initializeserationa data structure. See the excerpts below and the discussion above as to claim 1, discussingre, andthe initialization of an array, which inherently discloses the initialization of a data structure.	"Although MCode includes instructions and data structures needed to implement some Clarity language-specific constructs such as its exceptions and method calls, the core of MCode is suitable for representing code for C and many other languages." <i>Lewis</i> at 119.	AllocArray(array type index) ArrayIndex(array type index) ArrayLength(array type index) BeginExprCase(int) BeginDefaultCase()	InvokeOuter(method type index) InvokeDelegated(method type index) Widen(base interface number) Namow(target type index) DoContinue(end tag)	Figure 4: The MCode instruction set	Lewis at 123 Fig. 4 (excerpt).	"Figures 5 and 6 give an example of generated MCode. The Clarity method <i>startup</i> in Figure 5 produces the MCode instructions shown in part in Figure 6."
U.S. Patent No. 6,061,520 – C	U.S. Patent No. 6,061,520 - Claim 19	19. The computer-readable medium of claim 18 wherein the operation initializes a data structure, and						

right_sibiling: Worker, // right delegstes; manages "right_workers" workers };

Lewis in view of Dyer and further in view of Proebsting	iup: method (our_workers: in int) // executed when the thread is started; delegates most of its work to forked silving worklers within work, mutex { // acquire work, mutex for duration of the within statement	work_mutex	Threathenter		work_per_worker	our_work		extra_work	integer constant 0, type 0	extra_wort⊳0			our_work			integer constant 1, type 0	our_work∻=1	our_work		extra_work		integer constant 1, type 0	extra_work-=1	extra_work	iit(extra_wortk⊳D)(
Lewis in view of D	startup: method (our_workers: in int) { // executed when the thread is started; d within work, mutex { // acquire work, s	LosdGlobal(2)	invokeOuter(0x03010204)	our_work = work_per_worker;	LoadGlobag(1)	StoreGiobal(5)	if (extra_work > 0) {	LoadGlobai(3)	Losdint(0, 0)	Compareint(>)	SkipThen(cond_faise_tag_2)	$cur_work += 1;$	Giotral Addr(5)	Dup()	Loadindirect(0)	Loadint(0, 1)	Addint()	StoreIndirect(D)	extra_work -= 1;	GlobalAddr(3)	Bup() Eoseindirech(B)	LoadSigned(0, 1)	Subint()	Storeindirect(B)	SkipElse(end_if_tag_3)	BeginElse(cond_false_tag_2)	Endit(end_fl_ag_d)	,~~~.
U.S. Patent No. 6,061,520 - Claim 19																												

U.S. Patent No. 6,061,520 - Claim 19	Lewis in view of Dyer a	Lewis in view of Dyer and further in view of Proebsting
	LoadGlobai(2) InvokeCuter(0x02010205)	work_mutex Thread::exit
	delegated_to_workers: int = (our_workers - 1); EcadAm(0)	
	Losdint(0, 3)	integer constant 1, type 2
	Sutsint()	our_workers-?
	eo_work(our_work); LoaeGlobes(\$)	do work
	LoadGloba(5) Down24(7)	our_work An austrijner anne)
	ProcReturn(1)	method(our_workers
	Figure 6: MCode instructions generated	Figure 6: MCode instructions generated for the μC^{++} test program's startup method
	Lewis at 124-125, Figs. 5-6.	
	To the extent <i>Lewis</i> does not explicitly dis skill in the art would combine the disclosu	To the extent <i>Lewis</i> does not explicitly disclose this limitation of Claim 7, one of ordinary skill in the art would combine the disclosure of <i>Lewis</i> with <i>Dyer</i> to arrive at the claimed
	feature. Specifically, <i>Dyer</i> shows that an instruction for initializing a data struc array) was widely known in the art, as evidenced by its discussion of static array	a instruction for initializing a data structure (e.g., an denced by its discussion of static array
	Initialization:	
	"Mocha transformed a static initializer into an elegant, but illegal, construction:	o an elegant, but illegal, construction:
	public ConsoleWindow(String str	string, int il)
	= false; s = { "Plain",	};
	sizes = { "8", "9", "10", "12", "14", 	"l4", "l6", "l8", "24" };
	Bracketed initializer lists for arrays are va (either class or local), not for other assign	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure
	to me, but I'm sure Sun must have had a r	to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these

U.S. Patent No. 6,061,520 - Claim 19	Lewis in view of Dyer and further in view of Proebsting
	initializers are actually implemented by inline code inside constructors, generated by the compiler.
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." <i>Dyer</i> at 3.
wherein the simulating step includes the step of: simulating execution of the code to identify the initialization of the data structure.	<i>Lewis</i> discloses wherein the simulating step includes the step of: simulating execution of the code to identify the initialization of the data structure. See the excerpts below and the discussion above as to claim 1, disclosing the simulated execution of an array, which inherently discloses the simulated execution of a data structure.
	"The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." <i>Lewis</i> at 126.
	"In order to do high-quality code generation, we need to rebuild the expression trees from the stack machine. The code generator defers generation until the final target for an expression is known. Much of the process is similar to that of TNBIND algorithm documented in [Wulf 75]. This algorithm gives excellent results and executes extremely efficiently. A final optimization does instruction reordering to minimize RISC processor pipeline execution conflicts. By organizing the code generator into a series of cascading object streams, we are able to consume MCode and generate native machine code in one pass. Our object-oriented architecture provides an efficient way to trade increased memory for speed." <i>Lewis</i> at 127.
	To the extent <i>Lewis</i> does not explicitly disclose this limitation, one of ordinary skill would combine <i>Lewis</i> with <i>Dyer</i> and <i>Proebsting</i> to arrive at the claimed feature. Specifically, <i>Dyer</i> and <i>Proebsting</i> together show that the simulation of byte codes against a memory to identify underlying code, such as the static initialization of an array, was commonly known

U.S. Patent No. 6,061,520 - Claim 19	Lewis in view of Dyer and further in view of Proebsting
	to those of ordinary skill in the art at the time of the invention. For example, the decompiler technology disclosed by <i>Proebsting</i> discloses a technique for decompilation of Java byte code that simulates the Java virtual machine's stack:
	"This paper presents our technique for automatically decompiling Java bytecode into Java source." <i>Proebsting</i> at Abstract.
	" <i>Decompilation</i> transforms a low-level language into a high-level language. The Java Virtual machine (JVM) specifies a low-level bytecode language for a stack-based machine. This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a <i>class file</i> that contains type information and bytecode. The JVM requires a significant amount of type information from the class files for object linking Decompilation systems can exploit this type of information and well-behaved property to recover Java source code from the class file.
	We present a technique for transforming low-level Java bytecode into legal Java source code." <i>Proebsting</i> § 1.
	"We have implemented a prototype Java decompiler, Krakatoa, in Java." Proebsting § 5.
	"Krakatoa uses a stack simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations." <i>Proebsting</i> § 1.
	"Symbolic execution of the bytecode creates the corresponding Java source expressions Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." <i>Proebsting</i> § 2.
	Thus it would have been obvious to one of ordinary skill in the art to combine the methods of <i>Lewis</i> with the simulation (symbolic execution) of bytecode against a memory, as disclosed by <i>Proebsting</i> . And, even more specifically, the artisan would look to known decompilers,

U.S. Patent No. 6,061,520 - Claim 19	Lewis in view of Dyer and further in view of Proebsting
	such as those disclosed in <i>Dyer</i> . <i>Dyer</i> reviews three popular decompilers that were available before the '520 patent was filed: DejaVu, Mocha, and WingDis. <i>See Dyer</i> at 1. In that review, <i>Dyer</i> analyzes the decompilation of code, including the decompilation of static initializers:
	"Mocha transformed a static initializer into an elegant, but illegal, construction:
	W(String string, i "Bold", "Italic"
	sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." <i>Dyer</i> at 3.
U.S. Patent No. 6,061,520 - Claim 20	Lewis in view of Dyer and further in view of Proebsting
20. The computer-readable medium of claim 18 wherein the operation statically initializes an array and	<i>Lewis</i> discloses the computer-readable medium of claim 18 wherein the operation statically initializes an array.
אומוולמווללס מוו מוומן אווני	"Although MCode includes instructions and data structures needed to implement some Clarity language-specific constructs such as its exceptions and method calls, the core of MCode is suitable for representing code for C and many other languages." <i>Lewis</i> at 119.

Lewis in view of Dyer and further in view of Proebsting	BeginOwitch(end swi EndSwitch(end switc	BeginExprCase(int) BeginDefaultCase()	EndCasejend switch	DoBreak(end tag)	DoContinue(end tag)	Figure 4: The MCode instruction set		"Figures 5 and 6 give an example of generated MCode. The Clarity method <i>startup</i> in Figure 5 produces the MCode instructions shown in part in Figure 6."
	AllocArray(array type index) Arrayindexiarray type index)	ArrayLength(array type index)	InvokeOuter(method type index) InvokeDelegated(method type index)	Widen(base interface number)	Narrow(target type index)		Lewis at 123 Fig. 4 (excerpt).	"Figures 5 and 6 give an example of generated MCode. The Clarity Figure 5 produces the MCode instructions shown in part in Figure 6."
U.S. Patent No. 6,061,520 - Claim 20								

ž ž		our_work: int = 0; left_workers: int = 0; right_workers: int = 0; left_sibiling: Worker; // left delegatez; manages "left_workers" workers right_sibiling: Worker; // intit delegatez; manages "right_workers" workers	// the following declarations are private to the WorkerImpi implementation do_work: meitrod (work_ta_do: in int) { // elided */ };	<pre>inplement startup: method (our_workers: in int) {</pre>	Workkinpi: type = implementation of Worker	Worker: type = interface interface interface interface. // an unusual interface: no methods beyond Thread::startup and the other Thread nethods	U.S. Patent No. 6,061,520 – Claim 20 <i>Lewis</i> in view of <i>Dyer</i> and further in view of <i>Proebsting</i>	
			our_work: int = 0; ieft_workers: int = 2; right_workers: int = 2; ieft_sibing: Worker; right_sibing: Worker;	<pre> debegated_to_workers: if debegated_to_workers: if defegated_to_workers: a dete if (defegated_to_workers = dete if (left_workers = dete if (left_workers = dete if (left_workers = dete if (left_workers > 0)) night_sibling = new \ do_work(our_work); } for work(int = 0; ight_workers: int = 0; ight_workers: int = 0; ight_workers: int = 0; ight_workers: int = 0; ight_sibling: Worker; ight_sibling: Worker; ight_sibling: Worker;</pre>	inpleme inpleme fination fination fight work ison field site work field site work	Workimpi: typ impleme i i i i i i i i i i i i i i i i i i	<pre>work_mutex: work_per_wo extra_work:ik Worker:type % orkimpit typ % an unu % an unu % an unu % inpleme % inpleme % % finpleme % % finpleme % % % % % % % % % % % % % % % % % %</pre>	<pre>ThreadedSimulation: module work_marker: Threades: Muldex; work_marker: Threades: Muldex; work_marker: Threades: Muldex; work_marker: Threades: Muldex; work_marker: Threades: Threade</pre>
 If the following declarations are prodomers: method (work_to_do: in our_work: int = 0; is fight_workers: int = 2; is fight_workers: int = 2; is fight_workers: int = 2; is fight_sibiling: Worker; is fight_sibiling: Worker; is fight_sibiling: Worker; 	 // the following declarations are private to the WorkerImpi implementation do_work: method (work_to_do: in int) { /* elided */ }; our_work: int = 0; left_workers: int = 0; left_workers: int = 0; left_sibiling: Worker; // left delegates; manages "left_workers" workers right_eling: Worker; // right delegates; manages "left_workers" workers 	// the following declarations are private to the WorkerImpi implementation do_work: method (work_to_do: in int) { /* elided */ };		<pre>>> delegated_to_workers: int = (our_workers - 1); if (delegated_to_workers > 0) {</pre>	<pre>implement startup: method (our_workers: in int) {</pre>	<pre>Workimpl: type = implementation of Workers implement startup: method (our_workers: in int) implement startup: method (our_workers: in int) implement startup: method (our_workers: in int) implement startup: work, mutex for duration of the within statement within work_mutex i m acquire work, mutex for duration of the within statement implement our_work = work_per_worker; implement = work_per_workers = 0 {our_work = 1; extra_work = 1; implement = delegated to_workers = 0; implement = delegated to_workers = 0; implement = new Workers = 0; implement = new Worker = new W</pre>	<pre>ThreadedSimulation: module</pre>	ThreadedSimulation: module work_mutax: Threads: Midex; work_ret_worker: Int: // protected by work_mutex work_per_worker: int: // protected by work_mutex work_per_worker: int: // protected by work_mutex Workmpi: type = interface interface: no methods beyond Thread 0; // an unusual interface: no methods beyond Thread::stentup and the other Thread methods Workmpi: type = implementation of Worker // implement startup: method (our workers: in int) // implement startup: method (our workers: int) // implement startup: method
f do_work(our_work); // the following declarations are pr do_work: int = 0; ieff_workers: int = 2; right_workers: int = 2; right_workers: int = 2; right_sitting: Worker; right_sitting: Worker;	<pre>do_work(our_work); } do_work(our_work); } // the following declarations are private to the WorkerImpl implementation do_work: method (work_to_do: in int) { /* eided */ }; our_work: int = 0; ieft_workers: int = 0; ieft_workers: int = 0; ieft_eibing: Worker; if left delegates; manages 'left_workers' workers ight_eibing: Worker; if fight delegates; manages 'light_workers' workers </pre>	ر از the following declarations are private to the WorkerImpi implementation در work: method (work_to_do: in int) { /* exided */ };			<pre>inplement startup: method (our_workers: in int) {</pre>	<pre>Workimpl: type = implementation of Worker { inplement startup: method (our_workers: in int) in int)</pre>	ThreadedSimulation: module work_mutex: Threads: Mutex; work_per_worker: int: // protected by work_mutex work_per_worker: int: // protected by work_mutex work_mutex Warker: type = interface inherits Thread 0: // an unusual interface: no methods beyond Thread.::tartup and the other Thread nethods Workmpi: type = implementation of Worker // an unusual interface: no method cur_workers: in int) // # executed when the thread is stanted; delegates most of its work to forked sibling workers within work_mutex { // acquire work_mutex for duration of the within statement our_work = work_per_worker; // (extra_work > 0) {our_works + 1; extra_work - 1;) // (extra_work - 0) {our_works + 1; extra_work - 1;)	ThreadedSimulation: module work_mutex: Threads: Mutex; work_per_worker: Int. // protected by work_mutex work_per_work: int. // protected by work_mutex work_per_work: int. // protected by work_mutex Warker: type = interface: no methods beyond Thread::tarhup and the other Thread methods Workmpil: type = implementation of Worker implement startup: method (our_workers: in int) (
forktar: type // an unu // an unu // an unu // an unu // for // for // for for for for for for for for for for	<pre>Warker: type = interface interface: no methods beyond Thread::tertup and the other Thread methods // an unsuell interface: no methods beyond Thread::tertup and the other Thread methods // extens in the method (sur_workers: in int) // # secured when the thread as stathed; delegates most of the work to bread stathing workers within work, muset // accounted when the thread is stathed; delegates most of the work to bread stathing workers // # extens.work + 0) (our_work + = 1; // extens.work = 1; // delegated_10_workers = left_workers.]; // entry workers > 0; // entry workers.]; // entry workers > 0; // entry workers.]; // entry workers = left_workers.]; // entry workers.]; // fight_upointers = left_workers.]; // fight_upointers.]; // fight_upointers = left_workers.]; // fight_upointers = left_workers.]; // entry delegated_10_workers.]; // entry delegated_10_workers.]; // fight_delegates.]; // fight_delegates.]; // entry delegates.]; // entry delegates.]</pre>	<pre>Worker: type = interface interface: no methods beyond Thread::thartup and the other Thread methods</pre>	<pre>Warker: type = interface inherits.Threads::Thread::tartup and the other Thread methods // an unusual interface: no methods beyond Thread::tartup and the other Thread methods Working: type = implement tartup: method (our_workers: in it)</pre>	Worker: type = interface interface: Threads::Thread::Thread.0; // an unusual interface: no methods beyond Thread::startup and the other Thread methods WorkImpt: type = implementation of Worker {	Warker: type = interface interface interface interface; Thread::Thread 0; // an unusual interface: no methods beyond Thread::startup and the other Thread methods		ThreadedSimulaiton: module	ThreadedSimulation: module
off_mutex: off_mutex: forkimpt:type inpleme inpleme for do_work isitifent i	<pre>work_muter: Threads: Muter; work_muter: Threads: Muter; work_moder: Int; // protected by work_muter; work_moder: type = interface interins Threads: Thread: Thread of y work_muter; // an unusual interface: no methods beyond Thread: Thread interval; // an unusual interface: no methods beyond Thread: Thread interval; // an unusual interface: no methods beyond Thread: Thread interval; // an unusual interface: no methods beyond Thread: Thread interval; // an unusual interface: no methods beyond Thread: Thread interval; // an unusual interface: no work_muter; // an unusual interface: no work_muter; // a explore work_muter; // a explore work_muter (or work = 1;) // a context // a explore work_muter; // a context // a context</pre>	<pre>work_muter: Threads: Mutex; work_per_works:int.</pre>	<pre>work_mutex. Threads. Mutex; work_per_worker. Int: // protected by work_mutex work_per_worker. Int: // protected by work_mutex work_per_entergore. Int: // protected by work_mutex // an unusual interface. In methods heyond Thread.:sturup and the other Thread nethods // an unusual interface. In methods heyond Thread.:sturup and the other Thread nethods // executed when the thread is started; defegates most of its work to forked siting workers // implement startup. method (our_workers: in int) // executed when the thread is started; defegates most of its work to forked siting workers // implement startup. method (our_workers: in int) // executed when the thread is started; defegates most of its work to forked siting workers // implement startup. method (our_workers: int) // if (defegated_to_workers: if // acquire work_r= 1; exits_work = 1; // if (defegated_to_workers: if // acquire work.c= 1); // if (defegated_to_workers: if // workers: if // workers; // if (eff.workers = defegated_to_workers); // if (left.workers > 0) is f_siting = new Workerinnp(inght_workers); // if (left.workers > 0) is f_siting = new Workerinnp(inght_workers); // do_work(our_work); // do_work(int);</pre>	<pre>work_mutex. ThreadsMutex; work_per_worker: int;</pre>	<pre>work_mutex: Threads: Mutex;</pre>	.Mutex,		

Lewis in view of Dyer and further in view of Proebsting	startup: method (our_workers: in int) { // executed when the thread is started; defegates most of its work to forked sibling worklers within work_mutex { // acquire work_mutex for duration of the within statement	LosdGlobas(2) work_matex	ervertovers(preuder) Hardae.conce) Drift work = ventik Der ventiker	LosdGlobag(1) work_per_worker	StoreGiobai(5) our_work	$ii (extra_work > 0) ($	 Loadint(0, 0) integer constant 3, type 0	Compareint(≻) extra_work>0	20nd_fa\se_tag_2)	$cur_w or is += 1;$	Giobal Addr(5) our_work	Dup()	Loadindirect(0)	D, 43	Addint() Our_work+=1	Storeindirect(D) our_work	GlobalAddr(3) extra_work	Dup() Loadindireck(D)	LoadSigned(0, 1) integer constant 1, type 0	Storeindisect(0) extra_work	BeginEise(cond_false_tag_2) Endthernet # terr 21	
U.S. Patent No. 6,061,520 - Claim 20																						

U.S. Patent No. 6,061,520 - Claim 20	Lewis in view of Dyer a	Lewis in view of Dyer and further in view of Proebsting
	LoadGlobai(2) invokeCuter(0x02010205)	work_mutex Thread:rexit
	delegated_to_workers: int = (our_workers - 1); i cadaro(0)	
	Losaint(0, 1)	integer constant 1, type 2
	Sutsint()	our_workers-%
		ceregated_to_workers
	eo_work(our_work); •	
	LoadGlobai(5)	our work
		eo_workjour_work)
	8 ProcReturn(\$)	method(our_workers
	Figure 6: MCode instructions generate	Figure 6: MCode instructions generated for the μC^{++} test program's startup method
	Lewis at 124-125, Figs. 5-6.	
	To the extent <i>Lewis</i> does not explicitly dis skill in the art would combine the disclosu	To the extent <i>Lewis</i> does not explicitly disclose this limitation of Claim 8, one of ordinary skill in the art would combine the disclosure of <i>Lewis</i> with <i>Dyer</i> to arrive at the claimed feature.
	a use such that a potentially the subwe used an instruction for initialization of static array initialization:	idenced by its discussion of static array
	"Mocha transformed a static initializer into an elegant, but illegal, construction:	to an elegant, but illegal, construction:
	public ConsoleWindow(String str	string, int il)
	<pre>t dead = false; dead = false; styles = { "Plain", "Bold", "Italic" sizes = { "8", "9", "10", "12", "14",</pre>	"Italic" }; 12", "14", "16", "18", "24" };
	Bracketed initializer lists for arrays are va (either class or local), not for other assign to me, but I'm sure Sun must have had a r	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these

U.S. Patent No. 6,061,520 - Claim 20	Lewis in view of Dyer and further in view of Proebsting
	initializers are actually implemented by inline code inside constructors, generated by the compiler.
	When decompliing this same static initializer, WingDis produced equally beautiful and syntactically correct code." <i>Dyer</i> at 3.
wherein the simulating step includes the step of: simulating execution of the code to identify the static initialization of the array.	<i>Lewis</i> discloses wherein the simulating step includes the step of: simulating execution of the code to identify the static initialization of the array. See the excerpts below and the discussion above as to claim 1.
	"The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." <i>Lewis</i> at 126.
	"In order to do high-quality code generation, we need to rebuild the expression trees from the stack machine. The code generator defers generation until the final target for an expression is known. Much of the process is similar to that of TNBIND algorithm documented in [Wulf 75]. This algorithm gives excellent results and executes extremely efficiently. A final optimization does instruction reordering to minimize RISC processor pipeline execution conflicts. By organizing the code generator into a series of cascading object streams, we are able to consume MCode and generate native machine code in one pass. Our object-oriented architecture provides an efficient way to trade increased memory for speed." <i>Lewis</i> at 127.
	To the extent <i>Lewis</i> does not explicitly disclose this limitation, one of ordinary skill would combine <i>Lewis</i> with <i>Dyer</i> and <i>Proebsting</i> to arrive at the claimed feature. Specifically, <i>Dyer</i> and <i>Proebsting</i> together show that the simulation of byte codes against a memory to identify underlying code, such as the static initialization of an array, was commonly known

U.S. Patent No. 6,061,520 - Claim 20	Lewis in view of Dyer and further in view of Proebsting
	to those of ordinary skill in the art at the time of the invention. For example, the decompiler technology disclosed by <i>Proebsting</i> discloses a technique for decompilation of Java byte code that simulates the Java virtual machine's stack:
	"This paper presents our technique for automatically decompiling Java bytecode into Java source." <i>Proebsting</i> at Abstract.
	" <i>Decompilation</i> transforms a low-level language into a high-level language. The Java Virtual machine (JVM) specifies a low-level bytecode language for a stack-based machine. This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a <i>class file</i> that contains type information and bytecode. The JVM requires a significant amount of type information from the class files for object linking Decompilation systems can exploit this type of information and well-behaved property to recover Java source code from the class file.
	We present a technique for transforming low-level Java bytecode into legal Java source code." <i>Proebsting</i> § 1.
	"We have implemented a prototype Java decompiler, Krakatoa, in Java." Proebsting § 5.
	"Krakatoa uses a stack simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations." <i>Proebsting</i> § 1.
	"Symbolic execution of the bytecode creates the corresponding Java source expressions Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." <i>Proebsting</i> § 2.
	Thus it would have been obvious to one of ordinary skill in the art to combine the methods of <i>Lewis</i> with the simulation (symbolic execution) of bytecode against a memory, as disclosed by <i>Proebsting</i> . And, even more specifically, the artisan would look to known decompilers,

U.S. Patent No. 6,061,520 – Claim 20 U.S. Patent No. 6,061,520 – Claim 20 U.S. Patent No. 6,061,520 – Claim 21 21. The computer-readable medium of claim 18 further including the step of: running the created instruction on the	<pre>Lewis in view of Dyer and further in view of Pruebsting such as those disclosed in Dyer. Dyer reviews three popular decompilers that were available before the '520 patent was filed: DejaVu, Mocha, and WingDis. See Dyer at 1. In that review, Dyer analyzes the decompilation of code, including the decompilation of static initializers: "Mocha transformed a static initializer into an elegant, but illegal, construction: public ConsoleWindow(String string, int il)</pre>
processing component to perform the operation.	linkers or other tools." "Runtime generation of machine code offers many advantages. A runtime code generator can take advantage of information about the particular target platform to generate better

U.S. Patent No. 6,061,520 - Claim 21	Lewis in view of Dyer and further in view of Proebsting
	code. For example, different implementations of the SPARC architecture have different instruction pipeline properties. In the case of one new SPARC implementation, code generated specifically for this new processor can run up to 25% faster than code generated for a 'generic' SPARC processor. A runtime code generator can also take advantage of the specific values used in a program to generate machine code customized for those values. One of our goals is to explore the use of on-the-fly code generation for systems programming within Sun.
	Our representation of MCode object files is unusual in that we use platform-standard object files instead of some Clarity- or MCode-specific representation. This enables us to fully interoperate with existing C and SPARC ABI code. Object files containing MCode (which we call <i>Linkable MCode</i> files) are processed by standard linkers and other tools in the same way as other object files. As an example, our Solaris SPARC implementation supports complete interoperation with all SPARC Application Binary Interface (ABI) compliant code [SPARC ABI]. In particular, interpreted or compiled MCode programs can call C programs and vice versa, addresses of MCode procedures can be passed to C code and later called, and all C data types can be exchanged." <i>Lewis</i> at 119.
U.S. Patent No. 6,061,520 – Claim 22	Lewis in view of Dyer and further in view of Proebsting
22. The computer-readable medium of claim 18 further including the step of: interpreting the created instruction by a virtual machine to perform the	<i>Lewis</i> discloses the computer-readable medium of claim 18 further including the step of: interpreting the created instruction by a virtual machine to perform the operation. For example, <i>Lewis</i> discloses the implementation of SPARC instructions to execute the procedure's machine language entry code.
operation.	"While the MCode interpreter is mostly platform-independent, about 20% of its code is platform-specific. For example, in order to fully support procedure interposition and other ABI capabilities, the SPARC MCode interpreter does not directly interpret MCode ProcCall or Invoke instructions but instead implements them as SPARC ABI calls. Even MCode calls to other MCode procedures are implemented using SPARC instructions and execute the procedure's machine language entry code. This is necessary because the interpreter can

U.S. Patent No. 6,061,520 - Claim 22	Lewis in view of Dyer and further in view of Proebsting
	never know whether a called procedure is actually implemented in MCode or in C. (For example, a programmer might have replaced the called procedure using interposition.) This means the interpreter must fully handle all the details required for ABI calls. If a called routine will return an aggregate value, the interpreter must generate a sequence of machine instructions at runtime (a <i>thunk</i>) to support the SPARC ABI's calling convention that the returned aggregate's length must be encoded into a SPARC UNIMP instruction just after the call. The interpreter also stores all program values in memory as SPARC values since this is required for ABI interoperation.
	Recently, a second MCode interpreter has been developed by Mick Jordan. This interpreter executes <i>system models</i> written in the Clarity language. These system models precisely describe how a software system is built: the exact versions of its component parts, all options and build parameters, and how the component parts are assembled. This system modeller is intended to replace the Unix <i>make</i> tool and to eliminate some of its problems: e.g. the inability to exactly reproduce the construction of a software system. The system modeller's MCode interpreter is specialized to executing these models and to interacting with the Clarity program database. It does not need, for example, to support SPARC ABI interoperation." <i>Lewis</i> at 127-28.
	Figure 2: The runtime portion of the Clarity MCode system
	Lewis at 121.

U.S. Patent No. 6,061,520 - Claim 22	Lewis in view of Dyer and further in view of Proebsting
	This claim element would further have been obvious at the time of the invention to one of ordinary skill in the art from the teachings of <i>Lewis</i> , either by itself or in combination with other relevant prior art, including, but not limited to the <i>Dyer</i> and <i>Proebsting</i> references. <i>Dyer</i> is a review of various Java Decompilers. These are programs that convert Java class files into Java source code, effectively reverse engineering compiled code to figure out how the underlying code works. <i>Dyer</i> discloses, in most relevant part, an example of the code that would be used to implement a static initialization of an array. Applying one of the <i>Dyer</i> decompilers to, for example, Java code, would lead to the "[s]ymbolic execution [that] simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." <i>Proebsting</i> at § 2.
U.S. Patent No. 6,061,520 – Claim 23	Lewis in view of Dyer and further in view of Proebsting
23. The computer-readable medium of claim 18 wherein the operation has an effect on memory, and wherein the simulating step includes the step of: simulating execution of the code to identify the effect on the memory.	<i>Lewis</i> discloses the computer-readable medium of claim 18 wherein the operation has an effect on memory, and wherein the simulating step includes the step of: simulating execution of the code to identify the effect on the memory. For example, <i>Lewis</i> discloses that the concrete subclasses of CGValue represent the state of the individual entries on the simulated stack, a process which inherently identifies the effect of the play execution on the memory.
	"The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." <i>Lewis</i> at 126.
	To the extent <i>Lewis</i> does not explicitly disclose this limitation, one of ordinary skill would combine <i>Lewis</i> with <i>Dyer</i> and <i>Proebsting</i> to arrive at the claimed feature. Specifically,

U.S. Patent No. 6,061,520 - Claim 23	Lewis in view of Dyer and further in view of Proebsting
	<i>Dyer</i> and <i>Proebsting</i> together show that the simulation of byte codes against a memory to identify underlying code, such as the static initialization of an array, was commonly known to those of ordinary skill in the art at the time of the invention, and that it was used to identify the effect of an operation memory. For example, the decompiler technology disclosed by <i>Proebsting</i> discloses a technique for decompilation of Java byte code that simulates the Java virtual machine's stack:
	"This paper presents our technique for automatically decompiling Java bytecode into Java source." <i>Proebsting</i> at Abstract.
	" <i>Decompilation</i> transforms a low-level language into a high-level language. The Java Virtual machine (JVM) specifies a low-level bytecode language for a stack-based machine. This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a <i>class file</i> that contains type information and bytecode. The JVM requires a significant amount of type information from the class files for object linking Decompilation systems can exploit this type of information and well-behaved property to recover Java source code from the class file.
	We present a technique for transforming low-level Java bytecode into legal Java source code." <i>Proebsting</i> § 1.
	"We have implemented a prototype Java decompiler, Krakatoa, in Java." Proebsting § 5.
	"Krakatoa uses a stack simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations." <i>Proebsting</i> § 1.
	"Symbolic execution of the bytecode creates the corresponding Java source expressions Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." <i>Proebsting</i> § 2.

U.S. Patent No. 6,061,520 - Claim 23	Lewis in view of Dyer and further in view of Proebsting
	Thus it would have been obvious to one of ordinary skill in the art to combine the methods of <i>Lewis</i> with the simulation (symbolic execution) of bytecode against a memory, as disclosed by <i>Proebsting</i> . This would be for the purpose of "identify[ing] the [code's] effect on memory," as recited in the claim. Further, the artisan would look to known decompilers, such as those disclosed in <i>Dyer</i> . <i>Dyer</i> reviews three popular decompilers that were available before the '520 patent was filed: DejaVu, Mocha, and WingDis. <i>See Dyer</i> at 1. In that review, <i>Dyer</i> analyzes the decompilation of code, including the decompilation of static initializers:
	"Mocha transformed a static initializer into an elegant, but illegal, construction:
	<pre>public ConsoleWindow(String string, int i1) { dead = false; styles = { "Plain", "Bold", "Italic" }; sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };</pre>
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." <i>Dyer</i> at 3.