Dreams

R. J. Brown

Elijah Laboratories Inc.
201 West High Street
P. O. Box 833, Warsaw, Kentucky, 41095, USA

Abstract

Dreams is a message passing object oriented programming system. It allows for *before*, *after*, and *around* methods, multiple inheritance, and operator overloading. It is based on a prototype model of object oriented programming and dynamic scoping, rather than the more traditional class model and lexical scoping. Designed for real-time, Dreams has a fixed, predictable overhead.

A dream object is implemented as a contiguous data structure, and is therefore a suitable candidate for I/O. This makes it possible for objects and messages to be sent and recieved over a communications link or network, and for objects to be written to and read from a file or database.

Implementation techniques based on indirect threading, direct threading, token threading, and subroutine threading are discussed. The threading scheme chosen has a significant effect upon the efficiency of multitasking and MIMD parallelism. A hardware implementation of the dynamic binding mechanism used in Dreams is discussed. This hardware will permit determination of the currently active binding of any object without the addition of any processor cycles.

Introduction

The Dreams system provides a framework for a form of message passing object oriented programming. It makes use of a dynamic binding mechanism to permit the local view seen by a program to be temporarily altered. Within a dream, those words with local meaning to that dream have their execution behavior changed. When the dream finishes, the original meaning that was in effect at the dream's onset is restored.

We first discuss the application programmer's view of the Dreams system. Next we discuss the ways a systems programmer might go about implementing Dreams within a particular Forth programming system. Finally, we discuss ways to implement much of the underlying mechanism of Dreams in hardware, thereby eliminating most of the run-time overhead.

The inspiration for Dreams came from an experimental port of a Lisp Flavors package **[Moo86]**, and then a Lisp closures package **[Ric87]**, to Forth. After using this Forth closures package for a while, a number of significant changes were made that resulted in the current implementation of Dreams.

Dreams is currently written in LMI UR/FORTH **[Dun87]** and UR/FORTH-386 **[Dun91]**. Different Forth systems present different problems to the implementor of Dreams. A classical indirect threaded Forth interpreter, such as FIG-FORTH **[Tin89][Rag80]** and many FORTH-83 **[Dun85][Tin86]** systems use, requires a rather difficult approach. A direct threaded Forth, such as Zimmer's F-PC **[Tin89a]**, or Duncan's LMI UR/FORTH is quite well suited to a single tasking version of Dreams. A token threaded system is an excellent choice for a multitasking implementation of Dreams. A subroutine threaded system can be a good choice for a multitasking system if access to the compiler source code is available, but this approach can be difficult if the compiler optimizes some words into in-line code instead of subroutine calls.

Background

The advantages of an object oriented approach to software design have been widely discussed in the literature. Project Smalltalk at Xerox PARC was the first heavily funded investigation into the object oriented approach to software **[Gol83]**. The Department of Defense sanctioned language, Ada, has many provisions to support a form of object oriented software design **[ANS83][Boo83]**. The C++ extension to the C language provides object support for C **[Ell90][Eck90]**.

The Lisp language **[McC60]** has been the breeding ground for a number of object oriented approaches, including Closures **[Abe85]** Flavors **[Moo86]**, LOOPS **[Bob86]**, Object Lisp **[Dre85][Tel87][Tel89]**, and CLOS **[Gab89][Kee88][Ste90]**. The Whitewater Group's Actor is also an object oriented system **[Fra89]**. The X-Windows system **[Nye88]** and Microsoft's MS-Windows **[Pet88]** and Presentation Manager **[Pet89]** are all object oriented systems of a sort.

We have been developing an object oriented system called Dreams based on the Forth language. Although most currently popular object oriented programming systems operate with a class based object model, Dreams makes use of a prototype object model. The prototype model for objects is becoming more widespread, however. This is due to its simplicity. The SELF object oriented language **[Ung91]** uses prototypes.

Another Forth object oriented approach has been developed, but it uses an early binding strategy similar to Ada and C++ **[Pou87]**. TILE Forth is a portable Forth system written in C that runs under the unix operating system **[Pat90]**. TILE Forth is freely available under the Gnu Software License **[Sta87]**. It contains object oriented extensions for both class based and prototype based object models, thereby providing Forth-based examples of both approaches.

All this concern over object orientation is a result of a desire to make software easier to write, easier to re-use for other applications, easier to maintain, and generally easier to understand. Whereas conventional programming methodologies separate algorithm from data, the object oriented approach unifies the two with objects that behave in certain ways depending on what they are told and on what they already know.

The message passing practice of many object oriented systems is analogous to the way we as humans operate when we delegate a task to another person: we send them a message that tells them what to do. What they *actually* do is a function of what we told them, and what they already understand. In dreams, the messages are called thoughts. The objects are called dreams. They impose the understanding. Thoughts are interpreted, or "pondered", in the understanding of a dream.

Terminology

The terminology of Dreams (A glossary of Dreams terms is provided as an appendix.) is in parity with that of the Flavors **[Moo86]** OOP for the Symbolics Lisp Machines. The hardware of these machines is very similar to the hardware of Forth machines **[Sym84][Koo89]**. Lisp was a very strong influence on the design of the Forth language. Charles Moore, the inventor of Forth, took a graduate course in Lisp at MIT from John McCarthy himself, the inventor of Lisp, before starting to design Forth. (Personal conversation with Charles Moore, 1990.)

Flavors

The Flavors OOP is based on a metaphor that was inspired by an ice cream parlor near the MIT campus. Plain vanilla was the basic starting point for all of the store's confections. Many different mix-ins, such as sprinkles, nuts, fudge, fruit, etc. could be added to the vanilla and mixed in with it to produce new flavors. With only two dozen or so mix in additives, literally millions of flavors of ice cream deserts could be produced. This was the metaphor that inspired one of the most complete and widely used OOPs in the Lisp community. This metaphor accurately describes multiple inheritance in simple language anyone can understand. The base is plain vanilla, and the other flavors are mixins to the base.

This ice cream parlor is now a veritable shrine in the Lisp inner circle, somewhat like the National Radio Astronomy Observatory in Charlottesville, Virginia **[Tin89]** is to Forth. Visitors to ICAD, Inc. — vendors of an object oriented modeling system that is based on Flavors — make the pilgrimage from the ICAD training facility to the ice cream parlor on the last day of classes. When these newly indoctrinated students are told they are going out to get ice cream, they invariably ask, "Is this *the* ice cream Parlor?" They are answered with a softly whispered "yes," or maybe just a nod of the head. It is the high point of the week long training session.

Since we like Flavors very much — both in soft ice cream and in software — we used that package as a starting point in the design of what later became Dreams. This is appropriate: since Forth is Lisp inspired, a Forth OOP might well benefit from being likewise Lisp inspired.

But the Forth version of Flavors was clumsy to use. It was awkward to write to the Forth accustomed tongue. Remember, Forth is the only programming language, with the possible exception of COBOL, that was designed to be spoken as well as read and written. We needed to rethink the metaphor to fit Forth better.

Flavors is a late binding, lexically scoped, class based object oriented programming system. We viewed late binding as a requirement, since it provides a level of flexibility unobtainable with early-binding schemes, but the lexical scoping forced run-time look-up of methods, and that slowed down the implementation. Worse yet, it made the timing variable. Determinism was lost and performance was reduced from what we had desired.

Dreams

We eventually realized the significance of the fact that everything in a program is dynamic — changing with the passage of time as the program executes. This parallels real life as we live it. Each person has a slightly different perception of reality. A person has a different perception of his world when he is dreaming than when he is awake. In a dream, reality is warped, either slightly, or greatly, depending upon the dream. Much of a dream mirrors reality, but some things are not quite the same; they differ enough to make a dream world. When we are dreaming, we think we are awake, but we are not perceiving reality: we are perceiving the dream as though it were reality. We act out our lives in a dream, and experience the dream dynamically, just as we do reality. While we are dreaming, we cannot tell the difference.

We then considered recursive dreams — dreams that started not from the wakeful aware state of reality, but from within another dream. Here the warping is not with respect to reality, but with respect to the dream within which this other dream started. We saw great expressive power. We saw information hiding. We saw multiple inheritance. We saw polymorphism. We saw a new way for us to think about the OOP problem.

The dream is the object. Its purpose is to provide a way to warp the meanings of certain words, run some code (interpret the message), and then unwarp the meanings of the warped words. The result looks like a system based upon operator shadowing, since Forth is an operator language. Another view would say that it is a frame-based system where all the frames are related by exception links. The use of exception links in a frame based system of inheritance is not without theoretical difficulties. The difficulties of exceptions in inheritance based systems of representation are well explored by Brachman **[Bra85]**.

The new approach lent itself to a very clean implementation, and a fast, deterministic run-time execution. It is based upon the nesting of closures to shadow the bindings of selected operators. The inheritance is multiple, but unlike Flavors, which exhibits lexical scoping, the scoping in Dreams is dynamic.

Dreams uses dynamic scoping because a program is dynamic itself, and also because it permits a very run-time efficient and predictable implementation, as far as execution timing is concerned. The dynamic scoping seemed natural for the Dreams way of thinking, and the fall-out was that it can be made to run faster than lexical scoping, and that it is simpler to implement.

Lisp Machines Inc., the "other" Lisp machine company (besides Symbolics), proposed an OOP called Object-Lisp **[Tel89]** (pp 142-144)that was based on the same idea of nested closures and operator shadowing. They submitted the specification as a candidate for an OOP standard for Common Lisp **[Ste90]**, but this was defeated in favor of CLOS **[Kee88][Ste90]** instead.

Considerations for Real-Time

Although the advantages of an object oriented approach are well documented, there have been some very real problems making use of this approach in embedded real-time systems. Many object oriented systems, such as Smalltalk **[Gol83]** and Flavors **[Bro87]**, make use of sophisticated development environments that are difficult to eliminate after the software has been developed. Others, like Flavors, require extensive breadth first searches through a tree structured hierarchy of multiple inheritance to determine the applicable method to employ. Some require elaborate list structures and the overhead of garbage collection **[Ric87]**. Mark and sweep garbage collection causes a real-time system to go into a petite mal epileptic seizure: until the garbage collection finishes, no other processing can go on **[Knu68]** (pp 406-422,594). This is usually unacceptable. Ephemeral, or generation scavenging, garbage collection **[Jac90]** introduces shorter delays, but can still destroy deterministic timing.

Alternative attacks to this are seen in C++ and Ada. Both of these languages use object oriented techniques, but their power is somewhat lacking because they must determine at compile time which definition of an object is to be visible at run time. Some advanced techniques in object oriented programming cannot operate with this restriction.

To be applicable to a real-time environment, an object oriented system should be deployable on a minimal platform. To meet the demands of real-time operation, it should not make use of garbage collection or extensive method searching. It would be most desirable if the overhead to activate a particular object was fixed and determinable. To be truly flexible, it should not require that method determination be performed at compile time.

Dreams is written in Forth, and is therefore readily targetable to a wide range of deployment platforms. Forth is implemented on some of the largest and fastest computers in the world **[Dor86][Dor89]**, and also on some of the smallest and most humble microcontrollers available **[Dun86]**. It is the machine language of a new generation of RISC machines that are particularly well suited to real-time applications **[Dun89][Har88][Koo86]**. Due to Forth's ability to extend the compiler, the implementation of dreams is very straightforward, thereby easing portability and maintainability.

Dreams makes use of a dynamic binding mechanism that has a fixed and known overhead going into and coming out of a dream. Dynamic binding permits powerful late binding programming strategies to be used without the associated cost of run-time method searching overhead. Dreams was designed for real-time and written in a language that was designed for real-time. Forth is now directly executable as the machine language of several high performance RISC processors. At least one of these processors is available as a component of a standard cell library, thereby making incorporation of that processor into a custom integrated circuit quite straightforward and routine. This is the kind of deployment platform needed for many embedded applications.

In most cases, the overhead of dynamic binding should actually be less than the alternative overhead of basing all local instance variables on a pointer. The source code written using dreams will be devoid of cryptic pointer qualified variable references; it will use variables in the conventional manner. Indeed, dreams may even be used to retrofit existing code to a multiply instantiated application, even where that code's use of variables would normally prohibit its use in such applications. This is often the case when re-using code in a re-entrant multitasking application if that code was originally developed under a single-tasking, non-re-entrant assumption.

The Dreams System

A dream is the dynamic instantiation of a binding environment. In the context of a dream, certain symbols, represented as Forth words, have a different meaning, or effect, than they normally would have in the usual Forth sense. In the Dreams system, the underlying Forth environment — where all the Forth words have their usual meaning — is called Reality. When execution occurs in a dream, certain Forth words, either system defined, or user defined, take on a meaning peculiar to that dream, and potentially different from reality.

For those words that embody data storage, such as variables, a dream provides another data storage area that is local to the dream. This permits variables to have an existence in a dream that is different from that same variable in either reality or another dream. This allows dreams to be used to provide local variables, but it goes further than that. When local variables are maintained on the stack, they are created at the beginning of a routine, and destroyed at the end. For this reason, they must be initialized each time the routine is executed. Dream based variables provide for "instance variables". Instance variables continue their existence between one execution and the next. For this reason, they preserve local state. Dream variables can be used to remember values from one invocation of a routine to the next.

Dream variables could be used, for instance, to keep track of the cursor position in a text output routine. The cursor position would persist from the outputting of one character to the next. In a windowed display system, each window could be represented by a separate dream. Each dream could keep track of its own window's cursor position by the same named variables, but each dream would have its own private copy of them. Because the same variable names are used, but the data that those variables represent is different in each dream, a common set of low level words that reference those variables can be shared among all the window dreams.

Unlike most other object oriented approaches, where the object is a static entity that is manipulated by sending it messages that result in the application of methods to the object, dreams are consequences of the dynamic execution of a program. They only have a completely defined meaning while they are actually executing. The static data structure that holds the information needed to maintain instance variables and the alteration of word meanings is called the *essence* of that dream.

Dreams execute by pondering thoughts within the understanding of the dream. There are no specifically declared methods as such. Any Forth word may be pondered as a thought in a dream; however, any Forth word may take on a locally defined meaning within the understanding of a dream. If that word embodies data storage, the dream can provide a local private data storage area for use by that word. A dream can also provide an operator overloading capability that permits a word to have a locally different meaning within the understanding of the dream. For instance, a colon definition can be locally redefined within a dream to refer to a different sequence of threaded code than what that same definition refers to in the understanding of reality.

Consider again the windowed text output package alluded to above. If such a package had to operate with a range of different display terminals, the method for addressing the cursor would most likely be different for each terminal. We could have a different dream for each type of terminal, and have that dream provide a local definition for the cursor addressing routine as well as all other terminal dependent operations. When outputting to a particular window, that window's dream could have an over-loaded word for the type of terminal on which the window was displayed. This would permit the same code to display text for all windows on all terminals in the system without that code having to make any direct reference to what type of terminal it was being used on.

An Example

The appendix gives the advertised interface to the Forth words defined as part of the dreams package. Listing 1 gives a sample interactive session showing some exploratory programming using the dreams package. The reader should study the code and the results obtained, as it clearly demonstrates the different values that the same variables represent in the understanding of different dreams. It also shows how operator overloading of colon definitions is requested and the effects that this has.

First, three variables, **X**, **Y**, and **Z**, are defined and initialized. Then a word, **?**, to print a variable's value is defined. The word **DOT** is defined to print a value in a different format than **.** provides.

Instance Variables

The dream **snooze** is created with local instantiations of the variables **X** and **Y**. A copy of this dream is made and called **nap**. A thought that prints the variables **X**, **Y**, and **Z**, is defined and given the name **hmm**.

The thought **hmm** is pondered in reality to demonstrate its effect. It prints the values of **X**, **Y**, and **Z**, as they exist in the Forth system. This same thought is then pondered in the dream **snooze**, which produces the same effect: the same values of **X**, **Y**, and **Z** are displayed.

Now the values of **X**, **Y**, and **Z** are altered in the understanding of reality. When **hmm** is again pondered in reality, these new values are displayed, but when it is pondered in **snooze**, the old values of **X** and **Y** are displayed together with the new value of **Z**.

What has happened here? When **snooze** was created, it contained local versions of **X** and **Y**. The values of these variables were inherited from the values that they had when the copy was made, but **snooze** now has its own private versions of these variables. When **hmm** was first pondered in **snooze**, the original values were displayed. When the values of these variables were later changed, this change had no effect on the private copies of **X** and **Y** in **snooze**. Therefore, when **hmm** was again pondered in **snooze**, the old private values were still displayed. The new value of **Z** was displayed, however, because **Z** was not local to **snooze**, and so the same **Z** was known both to **snooze** and to reality.

An unnamed thought is then pondered in **snooze** to modify the values of the three variables within the understanding of that dream. When these variables are displayed in the understanding of **snooze**, the latest values are output, but when they are displayed in the understanding of reality, only the value of **Z** shows the change made in **snooze**. When the variables are displayed in the understanding of the dream **nap**, the values that were active when **nap** was copied are still active. All of these exercises serve to demonstrate the way dreams provide support for a form of local variables.

Classes

Now **FUE**, a progenator of dreams, is defined by the class defining word **TRANCE**. **FUE** defines a class of dreams in which the definition of **.** has local meaning. Since **.** is the Forth word to print a number, changing the definition of **.** will change the way numbers are printed. **FUE** is used to create two dreams, **FOO** and **BAR**. The thought **hmm** is pondered in each of these dreams to demonstrate that they both behave exactly the same as the understanding in which they were created.

Overloading

FOO is next told to imagine that the behavior of **.** is defined by the behavior of **DOT**. When **hmm** is pondered in **FOO**, the display of the variables is done according to the definition of **DOT**, with leading zeros and a trailing carriage return. The same thought is also pondered in **BAR**, which has not had its understanding of **.** changed. **BAR** displays the variables in the usual fashion. This demonstrates the locally defined operator overloading capability provided by dreams.

Multiple Inheritance

A thought is now pondered in **FOO**, but that thought causes the thought **hmm** to be pondered in **BAR**. Thus **hmm** is being pondered in **BAR** within the understanding of **FOO**. Although **FOO** has overloaded the definition of **.** with **DOT**, **BAR** still has the conventional definition of **.** in its understanding. This overloads the overload resulting in the conventional behavior of **.** being exhibited in spite of **FOO**'s different understanding.

FOO then ponders **hmm** within the understanding of **snooze**, causing the values of **snooze**'s version of the variables to be displayed using **FOO**'s version of the printing word. This shows multiple inheritance as a function of nested execution of dreams. The next example shows a doubly nested thought being pondered, resulting in a triply nested execution of a dream.

The dream **nap** is then pondered within the understanding of **FOO** to demonstrate that the overloaded behavior of **.** is inherited in the dream within a dream. The understanding at any given point is a result of the cumulative nested understandings of all dreams that are currently executing. This is one mechanism of inheritance in dreams. The other mechanism of inheritance occurs when the essence of a dream is created and the then active values and behaviors for all words local to that dream are copied over into the essence of the newly created dream.

Regression

Regression permits access to an understanding that was active when the current dream was invoked. In the next example, **FOO** ponders a thought that causes **hmm** to be pondered in the regression of **FOO**. Thus in this example, **hmm** is actually pondered in reality, since **FOO** was invoked from reality. Regression permits backwards navigation from a dream to the dream which that dream started in. The next example regresses back from **FOO** and then on into another dream, **nap**, where the thought **hmm** is pondered. This shows that the dream world may be visualized as having a tree structure, and thoughts may be pondered along forward, backward, and lateral pathways in that tree.

Early Binding

Early binding permits access to the definition of an individual word that was visible when a thought was compiled. Normally, the definition of a particular word is a function of the understanding in which the thought that contains a reference to it is pondered. This is late binding, or more particularly, since scoping is not lexical, dynamic binding. The word **REALLY** compiles a literal containing a copy of the pfa of the word following it. This is sufficient to reference a variable, as this is its address. To execute a colon definition that is early bound by **REALLY**, the word **DID** is required. **DID** executes a colon definition given its pfa in a manner analogous to the way that **EXECUTE** executes a colon definition given its cfa.

The thought **{ X ? }** pondered in **snooze** displays the value of **X** in the understanding of **snooze**, but the thought **{ REALLY X ? }** displays the value of **X** in the understanding of reality, since the thought was compiled in reality, even though it is pondered in **snooze**. This demonstrates early binding to variables. Early binding to colon definitions is demonstrated in the next two examples. When the thought `{ X @ . }` is pondered in **FOO**, the definition of **.** imagined in **FOO** is used to display the value of **X**, but when the thought `{ X @ REALLY . DID }` is pondered in **FOO**, the definition of **.** that was active in reality, when the thought was compiled, displays the value of **X** without the leading zeros.

Visions

Multiple inheritance occurs so frequently in advanced applications of object oriented programming that the dreams package provides a special construct for dealing with it explicitly — the vision. A vision is an ordered set of dreams. A thought may be seen in a vision the same way that a thought may be pondered in a dream. When a vision is activated, each of the dreams listed in the specification of the vision are activated in turn. The first dream specified has dominance and subsequently specified dreams have successively lesser influence over the understanding of the vision. When the vision is later deactivated, all the dreams are deactivated in reverse order. Complex factoring schemes, called inheritance networks, may be constructed by building a series of visions that share several dreams among themselves in different ways.

The next four lines of the example demonstrate the concept of visions. There are two visions, **BAZ** and **BOK**. They are both constructed from the same component dreams, but the dream ordering is different. When **hmm** is seen in **BAZ**, the definition of **.** in the understanding of **BAR** dominates, and the variables are displayed in the usual way, but when the same thought is seen in **BOK**, the definition imagined in the understanding of **FOO** dominates, and the variables are displayed with leading zeros and trailing carriage returns.

The next example demonstrates that a dream may be used as an argument to **SEE** in place of a vision. The effect is the same with respect to pondering the thought, only **SEE** incurs slightly more overhead than **PONDER**. When used this way, the dream is considered to be a vision with respect to lifting by **DEJA-VU**.

A nested vision, **BACH**, containing the dream **FOO** and the vision **BAZ**, is created and sent the thought **hmm**. Since **FOO** has precedence over **BAZ**, the imagined behavior of **.** predominates and the display of numbers appears with leading zeros and carriage returns.

Another nested vision is created with **BACH** as a component. The inheritance from **BACH** is overshadowed by the inheritance from **FOO** and **COMA**, but since **COMA** is the null vision, it will contribute nothing to the understanding of **HANDEL**. This demonstrates the ability of a vision to be composed of multiply nested dreams and visions.

Next, the vision **MOZART** is created from the dream **BAR** and the vision **HANDEL**. **HANDEL** inherits the leading zero behavior of **.** from **FOO**'s understanding, but this is overshadowed by the higher priority inheritance from **BAR**, which understands **.** in the normal sense. The result is that the numbers display without leading zeros when **hmm** is seen in **MOZART**.

Reality

The word **REALITY** regresses all the way back to reality, ponders a thought, and returns to the understanding active before the regression. This is demonstrated by sending **HANDEL** a thought that ponders **hmm** in **REALITY**. Normally **HANDEL** would have displayed the numbers with leading zeros, but the numbers are displayed without them in reality.

Dynamic Binding

Since the concept of dynamic binding is at the heart of any implementation of Dreams, we must discuss what dynamic binding is. Binding is a term that refers to the semantics, or action, associated with a name, or symbol. In Forth, we frequently use the term *word* to refer to the name of a subroutine, and also to the action of that subroutine, or the instructions that comprise it. For the purposes of this discussion, we shall use the terms *name* and *action*, and allow the term *word* to continue to have its customary meaning(s).

There are several different kinds of binding that need to be distinguished from each other: early binding, late binding, and dynamic binding. The closely related concept of scoping also comes in two varieties: lexical scoping, and dynamic scoping. Algol derived languages (the so called "structured" programming languages in vogue these days) operate according to the principles of early binding and lexical scoping.

Lexical scoping means that objects, such as variables, subroutines, etc., that are declared within a block of code are visible only within that block, and perhaps within blocks inside of that block if the particular language permits nesting of blocks. Ada, Modula, and Pascal do permit this, but C only permits a limited form of this, and FORTRAN does not permit it at all.

Early binding means, in this case, that it is the responsibility of the compiler, at compile time, to determine what action is associated with each name. The compiler compiles code that will produce this action when that code is later executed.

Since Forth is not generally considered to be a block structured programming language, we must reconsider what lexical scoping means in a Forth programming system. If we consider a word-list (which used to be called a vocabulary before the ANS X3J14 effort **[ANS91]**) to be the logical analog of the block in a block structured language, then lexical scoping and early binding means that the actions associated with the names in the current search order at the time of compilation will be compiled into the definitions of all new words. It should be apparent that this is just exactly what the Forth compiler does.

What about late binding? How does it differ from the early binding that we are familiar with? In late binding, the compiler does not compile code to *perform* the action associated with a name. It compiles code to *determine* the action associated with a name, followed by code to perform the action that has been determined.

Consider a late binding variation of the object oriented approach described by Pountain **[Pou87]**. In such a system, the compiler would compile code to search the list of methods belonging to an object, and when the name of the current word matched the name of the method, then the action associated with that method would be performed. This is known as run-time method searching, and is a characteristic of some late binding object oriented systems. Pountain realized that the overhead of method searching was too expensive for a real-time system, so he factored the method searching into compile time, resulting in an early binding implementation. This solved the speed problem, but made operations such as holding a message in a variable unfeasible. Late binding is needed if this is to be done.

So what is wrong with early binding? Why bother with a late binding strategy at all? In an early binding message passing object oriented system, the messages must be compiled at compile time, and the actions associated with the names of all the words comprising a message must be determined. This usually means that all the methods applicable to a particular object must be declared with that object at compile time. The messages that such an object may receive must be restricted so as to limit the words in the messages to only those words that have been declared valid for that object. In fact, the messages themselves are actually compiled as just more code in the sender's instruction stream, and the methods appear lexically following the recipient object. This is in violation of normal Forth usage, where an operator receives the data and acts on it. In this case, the message is the action, and the recipient of the message is actually passive. In an early bound system, it is not generally possible to have a variable hold a message, since the message needs to be compiled into the code that sends it to its recipient.

In a late binding system, a message can be an object too. A variable can hold a pointer to it. The message is passed passively on the stack to its recipient, which acts to interpret the message.

If the implementation uses run-time method searching, then each name in the message will be looked up and performed. This gives the applications programmer an additional flexibility over an early binding system.

In Dreams, a message is called a thought, and it is known by the execution token of any word, such as a colon-definition, variable, constant, etc. The execution token is passed on the stack to a dream that is to ponder that thought within its own understanding. While Dreams does not use an early binding approach, it does not use a method searching late binding approach either. Dreams uses a dynamic binding approach, which is a late binding technique with dynamic rather than lexical scoping.

The dynamic binding technique permits altering the action associated with a name. Since a name is known at execution time by the execution token compiled by the compiler from the name found in the source code, the dynamic binding implementation must rely upon knowledge of the inner workings of the compiler and interpreter. What is done in the current Dreams implementation is to provide a way to read and write the parameter field address of a word. This permits providing a different parameter field for a word local to a dream. Since the threaded code is deposited by the compiler into the parameter field, then by modifying the parameter field address, a word can be given an action at run time that is different from the action associated with it at compile time.

By reading the original parameter field address of a word and pushing it on an active bindings stack before writing the new parameter field address in its place, a means of restoring the original action is provided. Each dream has an associated list of execution tokens and substitute parameter field addresses that specifies that dream's contribution to its understanding. When a dream becomes active, the list of execution tokens and parameter field addresses is traversed. For each execution token in the list, the original parameter field address is read and pushed on the active bindings stack. After this, the new parameter field address, obtained from the list, is written in its place. When the dream finishes, the old parameter field addresses are restored by popping them from the active bindings stack. Since this list is of a fixed and known length, a fixed and known execution overhead results. This property of deterministic timing is generally desirable in many embedded and real-time systems applications.

An Implementation

The current Dreams project implements the Dreams system as an extension to LMI UR/Forth 1.03 **[Dun87]**. It has also been ported to LMI UR/Forth-386 **[Dun91]**. UR/Forth uses a direct threaded interpreter with a 16 bit cell size in the segmented real mode address space of an IBM-PC compatible running MS-DOS. UR/Forth-386 uses direct threading with a 32 bit cell size in the linear protect mode address space of an 80386 under MS-DOS with the Pharr-Lapp DOS extender. Listing 2 contains the source code for the UR/Forth Dreams implementation.

Direct Threading

The direct threading technique employed by UR/Forth compiles an execution token for each word that is the address of the native 8086 machine instruction to branch to in order to perform the action associated with that word. In the case of variables, the sequence of code loads the parameter field address into a register and then branches to the routine common to all variables. In the case of a colon definition, the sequence of code loads the parameter field address, which points to the compiled execution tokens for that word, into a register and branches to the code common to all colon definitions.

In both of these cases, the parameter field address is compiled into the immediate portion of a load immediate native machine instruction. This means that the address of the parameter field address may be determined from the execution token, which gives the code field address. This is fortunate, because we may always determine the address of the parameter field address from the execution token. Some similar situation should probably exist in other direct threaded Forth implementations.

In this Dreams implementation, the word **^pfa** converts the execution token of a word into a pointer to the parameter field address of that word. The words **pfa@** and **pfa!** fetch the parameter field address, and store a new parameter field address, respectively, when given the execution token of a word. These words are tailored to the UR/Forth model. If a different direct threaded Forth is to host Dreams, these words would need to be edited accordingly.

The words **new-bindings** and **old-bindings** are not advertised as part of the interface to Dreams. They are used to implement **PONDER**. The word **Find-Binding** is used by **IMAGINE** to look up the element in a Dreams list of local objects so that the local understanding of that object may be changed. This look-up only occurs when **IMAGINE** is executed, not every time that the dream is activated. Typically the use of **IMAGINE** is infrequent when a program is running. The words **bind-vision** and **unbind-vision** perform a function for visions analogous to the functions performed by **new-bindings** and **old-bindings** for individual dreams. All other words in Listing 2 have already been described.

Other Threading Techniques

When implementing Dreams on an existing Forth system, the implementor must provide some way of modifying either the code field address or the parameter field address associated with an execution token.

Indirect Threading

In indirect threading, such as is found in FIG-Forth **[Rag80][Tin89]** and numerous other Forths **[Dun85][Dun86][Tin86]**, the execution token is the address of the code field address. The parameter field address is typically the execution token plus one cell. These considerations make alteration of the code field address more practical than alteration of the parameter field address. A difficulty arises because the parameter field address is derived implicitly from the execution token, making it onerous to associate a separate parameter field with each new binding. If access to the source code for the Forth system compiler and inner interpreter is available, the implementer can change the implementation so that the parameter field is indirectly addressed like the code field.

Lacking this, all words that are intended to be rebound in a dream could be declared with special defining words, or these words could be given the names normally reserved for their nonrebindable counterparts. This last approach is only partially acceptable, because it does not permit rebinding any of the words that are already present in the kernel. This may be circumvented, albeit clumsily, by redefining each of these kernel words with the new defining words. This produces a new set of words with the same names and actions as the originals, except that they are repackaged to permit rebinding in a dream. Nonetheless, words that were compiled before the repackaging will not exhibit the expected altered behavior if they are pondered in a dream, since they were early bound when the kernel was compiled. The major disadvantage of such an approach is that extra overhead has been added to each and every word in the application.

Trampolines

Another approach to implementing Dreams in an indirect threaded Forth interpreter makes use of the concept of *trampolines*. Trampolines are short fragments of machine code that are created as needed and inserted into the otherwise normal sequence of machine code that supports a high level language function.

In an indirect threaded Forth system, the instruction pointer (IP) points to the execution token (ET). This execution token, in turn, points to the code field address (cfa). Typically, the IP is used to fetch the ET, and then the IP is incremented. The ET is used to fetch the machine instruction address to execute the Forth word associated with the ET. This machine code is typically the same for all words of the same kind, such as colon definitions.

If the behavior of a word's name is to be changed — if it is to be re-bound — the re-binding mechanism must fetch the cfa of the word, given its execution token. If this cfa does not point to a trampoline (which is determined by comparing to a skeleton trampoline), then a trampoline is allocated on the dictionary and the old cfa is pushed on the active bindings stack. The new trampoline is filled in to cause the newly bound action to occur instead of the old behavior. If the word already had a trampoline associated with it, then the trampoline's old machine instruction address would be pushed (just like a cfa) on the active bindings stack, and the new address would be stored in its place.

This technique has the advantage of not requiring modification to the Forth inner address interpreter. It also allows rebinding to different types of words, such as binding a variable name to the behavior of a colon definition. Its disadvantage is that once a word has been rebound, it will never be quite as fast, since the overhead of bouncing off of the trampoline will always be there once it has been inserted.

This overhead also exists in the unreclaimable storage associated with each trampoline. **FORGET** must never be used after any trampolines have been allocated, since a trampoline could be forgotten while its associated word was still in the dictionary. If such a word were to be executed, it would jump off to a trampoline that was no longer there. Great would be its fall!

Repackaging

The direct threading technique in the example exhibits no extra overhead during execution, but it has one shortcoming: a word may only be rebound to another word of the same type. This means colon definitions may only be rebound to other colon definitions; variables may only be rebound to other variables. Code words may not be re-bound at all. This makes it difficult to use before and after methods to place demons at watch over a variable, because the demons are colon definitions, and the variable may not be bound to colon definitions. These demons are necessary in order to implement *active values*. (Active values are variables that produce some extra action when they are referenced.) The kludge solution is to wrap the variable up as a colon definition that looks like:

```
VARIABLE my-var             \ define the variable
: my-var my-var ;           \ repackage it as a colon definition
```

which solves the problem in a way similar to the repackaging approach described for indirect threading above.

Both the direct and indirect threading implementations of Dreams produce an approximation to the ideal behavior. Indirect threading implementations will introduce extra overhead with every subroutine call. Direct threading implementations are restrictive in the kinds of rebinding that may be done: a word may only be rebound to the action associated with another word of the same kind, and code words may not be rebound at all. It is desirable to be able to rebind any word to any other word without regard to its type.

Multitasking

So far we have only considered single tasking situations. What happens in a multitasking situation? How do we perform a context switch when the very semantics of the words in the system need to be saved and restored? Since we have saved the old bindings of all altered words on the active bindings stack, we can unbind them back to their original behavior. A concurrent thread of execution must have, in addition to its own parameter and return stacks and instruction pointer, its own copy of each of the stacks in the Dreams implementation.

When a context switch occurs, the context switching mechanism must restore all the bindings pushed onto the old task's active bindings stack, and then rebind all the bindings implicated by all the dream essence pointers pushed on the new task's active essences stack. The first operation removes the understanding of the old task and restores the understanding of reality as a backdrop for the understanding of the new task. This is similar to regressing all the way back to reality. The second operation then recreates the understanding of the new task.

While this approach is semantically sound, and might serve for a user interaction application, it is generally far too slow for any demanding real-time situations. Because the active binding context is distributed all over the place in the various modified parameter field addresses or code field addresses, it is a very difficult thing to quickly save and restore. The slow and laborious process of unbinding each altered behavior of the old task one by one and restoring each altered behavior of the new task one by one seems to be the only way to implement a context switch under these threading schemes.

Token Threading

With a token threaded Forth, each execution token is an offset into a table of code field addresses. The inner interpreter fetches the execution token pointed to by the instruction pointer and indexes into the token table, where it fetches the code field address of the word. Parameter fields may be addressed in various ways, depending upon the details of the implementation. One approach would be to have any word needing a parameter field load its address with a load immediate instruction in a manner similar to the direct threading scheme. In this case, it is more convenient to alter the code field address of a word, and have the new code field provide a new parameter field if necessary.

The beauty of the token threaded approach is that all of the altered bindings are conveniently contained in a single table. Each task can be provided with a save buffer to hold its version of the altered token table. To perform a context switch, we can merely copy the token table off to the old task's token save buffer, and copy the new task's save buffer into the token table.

If we have access to the source code for the inner interpreter, we can even do better than that: we can have the inner interpreter find the token table by an active token table pointer. This eliminates all the copying and provides a context switch that only requires that we swap pointers to the token table rather than swap the contents of the table. On a machine with a double indexing capability, the active token table pointer can be held in an index register, thereby reducing interpretation overhead to a minimum. If double indexed indirect addressing is available, the entire inner interpreter may be reduced to a couple of instructions. If post-incrementing and chained indirection are also available, the inner interpreter might even be implemented as a single instruction.

Subroutine Threading

In Forth systems that use subroutine threading, a form of tokenization can be implemented by using doubly indexed indirect addressing so that subroutine calls are vectored through the active task's token table. If the compiler optimizes certain words as inline native instructions instead of subroutine calls, this will present a problem if it is desired to alter the behavior of these words by dynamic binding. If access to the compiler source code is possible, the words can be implemented as subroutines also, thereby eliminating the problem; otherwise, it may be possible to repackage these words in the manner described above for indirect threaded systems. If the compiler is particularly good at optimization, it may be smart enough to desubroutinize these repackagings and thwart our efforts. In this case, the implementation may have to forgo complete freedom of dynamic binding and restrict these words from being rebound.

Hardware Assistance

We have devised a hardware implementation of the dynamic binding mechanism used in Dreams. This hardware technique permits determination of the currently active binding of any execution token without the addition of any processor clock cycles. Since the establishment of a new binding only involves a few instructions on a processor equipped with the Dreams hardware, and the use of any such binding is absolutely free with respect to CPU time, this invention provides real-time embedded systems with a very effective way to incorporate object oriented programming. This invention can switch the binding environment to a totally different environment in only one machine cycle. This is advantageous in multi-tasking interrupt driven embedded real-time systems.

For example, the Harris RTX-2000 series of Forth-based RISC microcontrollers can service an interrupt by performing a context switch on the processor registers in 400 ns, which is 4 machine cycles [Har88][Har90]. With the Dreams hardware, only 1 additional machine cycle is needed to perform a context switch, including the switching of the complete active binding environment. To perform a complete context switch between two processes in an object oriented program in only 5 machine cycles, or 500 nanoseconds, is impressive.

The hardware itself is composed of a bank of fast static RAM inserted into the instruction fetch logic of the processor. This RAM performs a programmable address translation on the destination address of each subroutine call. This RAM token translation table, or TTT, is divided into pages, with one page for each task. When a task switch occurs, the active TTT is changed by switching the active page in RAM. Since the processor uses a significant portion of a clock cycle to determine the type of an instruction, the TTT can translate the destination portion of a subroutine call instruction into a new destination by addressing the TTT with the destination of the original subroutine instruction and using the contents fetched from the TTT as the actual subroutine address. This translated address is then placed into the program counter to determine the next instruction address. The translation is performed in parallel with the instruction decode, so no time penalty is paid.

To change the binding of an execution token, all that is needed is to store the new binding address into the associated token slot in the TTT. Of course, to support the Dreams system, the old binding must be read and pushed onto the active bindings stack, but this is a detail that is handled by the software in the Dreams programming system. To perform a task switch, the bindings active in the old task must be saved, and the new task's bindings must be restored. This is done very quickly by having enough pages in the RAM to support all the time-critical tasks in the application, and at task switch time, performing an output operation to select the active page of the RAM. This saves the old bindings and restores the new ones in a single machine cycle. Typical embedded applications should not use over 1 K tokens in RAM per task. The RAM must be fast enough to keep up with the processor instruction decode operation. The TTT must be wide enough to hold a destination address. On the RTX-2000, this is 15 bits for the small memory model, and 19 bits for the large memory model.

On the RTX-2000, the machine cycle is 100 ns, and 25 ns static RAM would do the job just fine. Harris has indicated (Personal conversation with Harris marketing representative, 1989.) that with the RAM on the same chip as the processor, 40 ns RAM would do. They say it is straightforward to put 8 K cells of such RAM on the same chip as the processor. This would support 8 tasks with a 500 ns context switching overhead running an object oriented program. The cost for a commercial grade chip should be around $65.00 in quantity. We do not feel that there is another processor around that can come anywhere near this price/performance ratio for a real-time micro-controller running object oriented applications.

The same ideas are readily adaptable to the John Hopkins University Applied Physics Laboratory Forth-on-a-chip project **[Hay89]**, and Phil Koopman's WISC machine **[Koo86][Koo89]**. The WISC has a writable instruction set, so the dynamic binding mechanism could be implemented with writable firmware, which means the additional hardware is not needed, but the execution time would be a bit slower.

Future Research

The representation of the essence of a dream as a contiguous region of memory was a deliberate design consideration for a number of reasons. The defining word for a dream has the disadvantage, in its current form, of requiring the programmer to name those words the dream is closed over. Only words whose names were declared when the dream was instantiated may have their bindings altered in that dream.

Dynamic Reslotting

An alternative approach would only require the programmer to specify the number of rebindable slots the dream should contain, with the system providing a reasonable default value. In this approach, all of the slots would contain zero. The rebinding mechanism would not loop through all the bindings, but only as far as a slots-used counter indicated. All slots containing a value of zero would be skipped over. Additional Dreams words would allow the dynamic insertion and deletion of word names from the closure list of a dream.

A problem arises here, however: what would happen if a dream reslotted itself while it was active? The unbinding mechanism would become confused and not restore the old bindings properly. The same thing could happen if some other task altered the binding slots of a dream while it was active. Note that this is not the same thing as if the *behavior* associated with a name is altered while the dream is active. This is a fine thing to do, and will cause no such confusion when restoring old bindings. It is necessary to realize, however, that the new binding specification will not become active until the dream is activated *again*, after the new binding has been specified. But when the *name* of a word to be closed over, not the behavior to bind such a name to, is altered, the old binding saved on the active bindings stack will get stored into the wrong memory location at unbinding time. Such a catastrophe should be avoided. The idea of run-time changes to the closed over name list of a dream is attractive, but some efficient means must be found to detect whether a dream is active before such a scheme would be very robust.

Another solution would be to keep both the old bound value and its execution token on the active bindings stack and to not look at the essence at all during the restoring of the old bindings. Although this would use twice the space on the active binding stack, it would provide a robust solution to the problem.

Information Hiding

Another area that needs to be explored is information hiding. Dreams currently provides no mechanism for information hiding, as is present in most other object oriented programming systems. The Forth word-list (vocabulary) capability provides control of the Forth name space. If a defining word for a dream somehow created a corresponding word-list associated with that dream, then words private to that dream could be put into its word-list, and publicly visible words could be put into the application's word-list. We suspect that good use of this facility in a higher level set of defining words for dreams would be able to provide the information hiding features that are currently lacking from Dreams.

Control Delimiters

We are currently developing a portable unix based Forth system written in C that is designed around a token threading scheme. It is designed to run Dreams efficiently even when cooperative multitasking Forth applications are being executed. This system is also designed to permit exploring several novel control constructs. Exception handling in ways similar to PL/1, Ada, Lisp and Scheme have caused us to consider an efficient way to implement these *non-local exits* in a Forth system **[Bro90]**. This opens the way to a complete Forth implementation of *control delimiters* **[Sit90]**. These are very powerful concepts. It is amazing how hard it is to implement them in most languages (other than Lisp), but after developing the tree-structured stack concept, all of these control structures are almost obvious.

Debugging

Another nice capability, from an incremental development and debugging viewpoint, is that with these tree-structured stacks, a thread of execution spun off of another thread immediately inherits the entire stack of its parent thread. This is done with no copying or other overhead. In actuality, the stack structure is the same for a subroutine call as it is for a thread spawning operation. The only difference is that a subroutine call waits for the subroutine to return before it proceeds, but the parent thread executes in parallel with the spawned thread.

Because of this structure, a breakpoint or other error can transfer control to a debugger, where the programmer can look at data and code, etc. The code can be edited to fix the problem. At this point, using a conventional Forth system, the programmer would have to reload the program and rerun it from the beginning, hoping that the problem did not recur. One of the advantages of Forth over other programming languages has always been the ease with which the programmer could do these steps, but if the program takes a very long time to run until it gets to the point of the error, a language that compiles slower but executes faster (such as C) would have an advantage.

Rewinding Execution

In the Forth system we are developing, because of the tree-stack architecture, only the edited routine needs to be recompiled. This is the first time-savings. Then, instead of starting execution over at the beginning of the program, the program is *rewound*, so to speak, to the point where that subroutine was called. Because of the tree-structured stacks, the status of the stacks is reclaimable at the point of the call. This permits execution to do a "take two", as it were, on that subroutine, without having to run all the rest of the program over. This is only possible, however, if no global variables have been modified. This includes dream-local instance variables. If the only variable modifications have been to variables on the stacks, this rewinding can occur. This can save a tremendous amount of time.

Further more, if the bug showed up during a real-time run, running the program again cannot guarantee that the same bug will show up twice, but if the same program is rewound a little bit and played again, all the data values collected in real-time will still be there, and the situation that revealed the bug will occur again. This can represent a tremendous savings of the programmer's time and of development expense.

Dreams and Tree-Stacks

But to realize this powerful development environment, Dreams and the tree-stacks will have to co-exist in the same Forth system. This presents difficulties more profound than one would imagine. When the control environment changes due to a multi-tasking context switch, we have seen how to handle the Dreams object binding environment, but what happens when a **THROW** is caught by an earlier **CATCH**? The status of the binding environment must have been saved at the **CATCH** in order for the **THROW** to properly restore it. A brute force solution would be to save a copy of the token translation table for every **CATCH** or other such control point. This would rapidly get out of hand, both for the time to make the copy, and for the memory that it would consume.

A central point to the architecture of our Dreams Forth system being developed is a mechanism to keep track of this binding environment along with the stacks associated with it for every flow of control that is active. The structure that results is similar to the shallow binding mechanism used in many Lisp systems [All78] (Sec.3.11, pp 149-153). It should still permit the hardware speed-up by using a hardware token translation table, and also the hardware implementation of the tree-structured stacks. We are now looking into the feasibility of implementing the shallow binding mechanism in a programmable gate array.

Message I/O

A message is a contiguous region of memory, being a colon definition. This permits messages to be transmitted and received, or written to mass storage and read back at a later time.

This only works, however, if the addresses in the dictionary are the same when the message is input as they were when the message was output. If the addresses are different, assuming an address threaded Forth, such as indirect or direct threading, this scheme fails.

If a token threading system is used, the only requirement is that the same tokens be bound to the same behaviour, but if the behaviour that a token is bound to is different, the metaphor is still good: it is just that a different mechine may provide a different view of reality — it is really a different dream.

If these requirements are not met, or even if different machines altogether are used, the message can still be transmitted by decompiling it back to source format (or storing it as a source text character string), and then sending it to the remote machine where the compiler on that machine can produce a representation of the message intelligible to *that* Forth system. Remember, a message is a command, expressed in Forth, either as source text or in compiled form.

Object I/O

Dreams was deliberately designed with a representation that uses a single contiguous region of memory to store the essence of a dream. This allows the essence to act as a buffer for I/O operations. This way, a dream may be sent over a communications link to another processor, or written out to a mass storage device to implement a persistent object.

The only problem with this, using the implementation described earlier, is that addressing of dream local instances will fail if the dream is not input back at the same address from which it was written. The solution to this is to use a position independent representation, such as IP relative addressing, or addressing by an offset from the base address of the essence of the dream. This should not be too dificult to implement, especially if the Forth system is designed for position independent coding to begin with.

Migrant Workers

A particularly interesting idea is to have the context of a thread of execution be contained inside of a dream, using local instance variables. This would allow multitasking by juggling the execution of different dreams. One could even swap a dream out to mass storage and swap it back in later, or send a dream to another processor, implimenting a sort of vagabond task, or migrant worker.

The use of such migrant workers holds promise as a representational scheme in applications such as factory automation and CIM. In these applications, a peice of work typically moves from machine to machine to have different operations performed on it, such as drilling, milling, welding, deburring, anealing, etc. A migrant worker task could be associated with each such object, and the task would travel with the object, from machine to machine, as the manufacturing progressed.

MIMD Paralellism

The I/O capabilities of messages and dreams make them ideal to implement MIMD (Multiple Instruction Multiple Data, as opposed to SIMD — Single Instruction Single Data.) parallelism **[Hil85]**. Each processor can communicate with other processors by sending messages. Objects can likewise be exchanced between processors. Even tasks can travel across the network between processors. Multiple processor systems can exist over local area networks, tightly coupled shared memory backplanes, or even wide area switched networks, such as the public telephone system. All of these architectures could benefit from the cohesive object oriented representation used in Dreams. Remember, position independent token threaded compiled code is portable across a network even if non-homogeneous processors are deployed.

Summary

The Dreams object oriented system has been described and demonstrated as a message passing object oriented extension to the Forth language. It is rather unusual as object oriented systems go since it uses a dynamic binding technique. The advantages of dynamic binding are that it permits determination of object binding at run time instead of compile time, and that it eliminates the need for run time method searching and the associated run time overhead. These advantages make Dreams a viable candidate for real-time object oriented applications.

Several types of binding and scoping have been described and differentiated. Dynamic binding has been described and contrasted to early binding and to method searching late binding. The importance of dynamic binding in implementing the Dreams system has been brought out.

Techniques for implementing Dreams in various Forth systems employing different threading strategies have been elaborated upon, with emphasis placed on what restrictions might apply, and the efficiency of the implementation.

Multitasking introduces the need for a context switching mechanism, and the context in a Dreams based application includes all of the altered bindings currently active in each of the tasks. The complications of context switching in a dynamically bound multitasking environment have been exposed and the efficiency of such a context switch studied for different implementations.

An example of a working implementation of Dreams has been discussed, and source code for the implementation has been provided.

A description of a hardware implementation of the dynamic binding mechanism used by Dreams has been presented. This hardware appears viable for embedded systems applications that need maximum performance in multitasking situations. The added hardware for typical embedded systems could probably be placed right on the chip with the Forth engine and stack controller. If larger binding environments need to be supported, more expensive off-chip implementations could be developed.

Acknowledgements

The Dreams concept was inspired by a port to Forth that I made of a Lisp Flavors and Closures package **[Ric87]** written by Pete Ohler of Boulder Creek, California. This Closures package was written in muLISP-87. The muLISP package was itself ported from the dynamic Closures package for the Lisp Machine project at MIT. Because of the unique syntactic and semantic nature of Forth, the closures package was not initially a good match to Forth, but after numerous changes were made as experience was gained working with the Forth closures package, the metaphor of a dream was conceived. With a consistent metaphor to guide the design of a new but similar package, a good match to the style of Forth was obtained. I would like to thank Pete Ohler for providing the inspiration for Dreams.

I want to thank Gregory Ilg for providing a sounding board during the evolution of the Dreams system. He was the first test case for any metaphorical innovations, and he shot holes in many of them.

I would like to thank the following people who read the drafts of this paper and offered many suggestions and corrections: Scott Isemenger, Kevin McArthur, Trish Renner, Ron Strausser, and Scott Thompson. I especially want to thank my wife Darlene Brown for proof reading the drafts of this paper many times.

I would like to thank the referees for their careful reading of this paper, and their many suggestions for its improvement. This is a better paper because of them. Nevertheless, all errors must be my own.

Availability

The source files for Dreams are available for downloading from the Elijah Laboratories, Inc. customer support bulletin board system as the file DREAMS.ZIP. The modem phone number is: (606) 567-2102. Modem parameters are 2400 Baud, 1 stop bit, 8 data bits, no parity, xmodem, ymodem, or zmodem protocol.

References

[Abe85] Abelson, Harold & Sussman, Gerald. 1985. Structure and Interpretation of Computer Programs. MIT Press, Cambridge, Massachusetts.

[All78] Allen, John. 1978. Anatomy of LISP. McGraw-Hill, New York.

[ANS83] ANSI/MIL-STD-1815A. 1983. MILITARY STANDARD Ada Programming Language. Department of Defense, Washington D.C.

[ANS91] X3J14 dpANS-2-Aug 3, 1991. draft proposed American National Standard for Information Systems --- Programming Languages --- Forth. Computer and Business Equipment Manufacturers Association (CBEMA), Washington, DC.

[Bob86] Bobrow, D. G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., & Zdybel, F. 1986. "CommonLoops: Merging Lisp and Object Oriented Programming." In OOPSLA '86 Conference Proceedings. Published as SIGPLAN Notices, Vol 21 No 11 pp 17-29.

[Boo83] Booch, Grady. 1983. Software Engineering with Ada. The Benjamin/Cummings Publishing Company, Menlo Park, California.

[Bra85] Brachman, Ronald J., 1985. "I Lied about the Trees, or Defaults and Definitions in Knowledge Representation." The AI Magazine. American Association for Artificial Intelligence, Melno Park, California, Fall 1985, Vol 6, No 3, pp 80-93.

[Bro87] Bromley, Hank & Lamson, Richard. 1987. Lisp Lore: A Guide to Programming the Lisp Machine, 2nd ed. Kluwer Academic Publishers, Boston, Massachusetts.

[Bro90] Brown, R. J. 1990. "Non-Local Exits and Stacks Implemented as Trees." Proc. 1990 Rochester Forth Conference, Rochester, New York.

[Dor86] Dorband, John E. 1986. "MPP Parallel Forth." In Frontiers of Massively Parallel Scientific Computation. NASA Conference Publication 2478. NASA Scientific and Technical Information Office. pp 275-283.

[Dor89] Dorband, John E. 1989. "Parallel Forth." The Journal of Forth Application and Research. Institute for Applied Forth Research, Rochester, New York. Vol 5 No 4 pp 459-468.

[Dre85] Dresher, G. 1985. Object Lisp . Lisp Machines, Inc. Cambridge, Massachusetts.

[Dun85] Duncan, Ray. 1985. PC/Forth Language Reference Manual. Laboratory Microsystems Inc. Marina del Rey, California.

[Dun86] Duncan, Ray. 1986. LMI Forth-83 Metacompiler Version 2.1 Reference Manual. Laboratory Microsystems Inc. Marina del Rey, California.

[Dun87] Duncan, Ray. 1987. UR/FORTH User's Manual, Laboratory Microsystems Inc. Marina del Rey, California.

[Dun89] Duncan, Ray. 1989. "A New Breed of Microcontroller." Embedded Systems Programming. Miller Freeman Publications, San Francisco, California. Vol 2 No 2 pp 15-18.

[Dun91] Duncan, Ray. 1991. 80386 UR/FORTH User's Manual. ver 1.16. Laboratory Microsystems Inc. Marina del Rey, California.

[Eck90] Eckel, Bruce. 1990. "C++ for Embedded Systems." Embedded Systems Programming. Miller Freeman Publications, San Francisco, California. Vol 3 No 1 pp 36-48.

[Ell90] Ellis, Margaret A. & Stroustrup, Bjarne. 1990. The Annotated C++ Reference Manual. Addison-Wesley, Reading, Massachusetts.

[Fra89] Franz, Marty. 1989. "Writing Filters in an Object-Oriented Language." Dr. Dobbs Journal. M Publishing Co. Redwood City, California. Vol 14 No 12 pp 28-36.

[Gab89] Gabriel, Richard P. 1989. "The Common LISP Object System". AI Expert. Miller Freeman Publications, San Francisco, California. Vol 4 No 3 pp 54-65.

[Gol83] Goldberg, Adele. 1983. Smalltalk-80: the language and its implementation. Addison-Wesley, Reading, Massachusetts.

[Har88] Harris Corporation. 1988. Harris RTX-2000 Programmer's Reference Manual. Harris Corporation, Semiconductor Sector, Melbourne, Florida.

[Har90] Harris Corporation. 1990. The RTX 2000 Hardware Reference Manual. Harris Corporation, Semiconductor Sector, Melbourne, Florida.

[Hay89] Hayes, John & Lee, Susan. 1989. "The Architecture of the SC32 Forth Engine." The Journal of Forth Application and Research. Institute for Applied Forth Research, Rochester, New York. Vol 5 No 4 pp 493-506.

[Hil85] Hillis, W. Daniel. 1985. The Connection Machine. MIT Press. Cambridge, Massachusetts.

[Jac90] Jackson, Frank. 1990. Generation Scavenging. Dr. Dobb's Journal. M Publishing Co. Redwood City, California. Vol 15 No 5 pp 16-28.

[Kee88] Keene, Sonya E. 1988. Object-Oriented Programming in COMMON LISP: A Programmer's Guide to CLOS. Addison-Wesley, Reading, Massachusetts.

[Knu68] Knuth, Donald E. 1968. The Art of Computater Programming, Vol 1 "Fundamental Algorithms." Addison-Wesley, Reading, Massachusetts.

[Koo86] Koopman, Phil Jr. 1986. "MVP Microcoded CPU/16 Architecture." Proceedings of the 1986 Rochester Forth Conference. The Journal of Forth Application and Research. Institute for Applied Forth Research, Rochester, New York. Vol 4 No 2 pp 277-280.

[Koo89] Koopman, Phil Jr. 1989. Stack Computers: the New Wave. Ellis Horwood Limited Chinchester, West Sussex, England.

[McC60] McCarthy, J. 1960. Recursive functions of symbolic expressions and their computation by machine. CACM, Apr. 1960, pp 184--195.

[Moo86] Moon, D. A. 1986. "Object-Oriented Programming with Flavors." In OOPSLA '86 Conference Proceedings. Published as SIGPLAN Notices, Vol 21 No 11 pp 1-16.

[Nye88] Nye, Adrian. 1988. Xlib Reference Manual. O'Reilly Associates, Inc, Newton, Massachusetts.

[Pat90] Patel, Mikael R. K. 1990. TILE man pages. CADLAB, Dept. Computer Science, Linkoping University, S-581 83 Linkoping Sweeden. mip@ida.liu.se

[Pet88] Petzold, Charles. 1988. Programming Windows. Microsoft Press, Redmond, Washington.

[Pet89] Petzold, Charles. 1989. Programming The OS/2 Presentation Manager. Microsoft Press, Redmond, Washington.

[Pou87] Pountain, Dick. 1987. Object Oriented Forth: Implementation of Data Structures. Academic Press, London, England.

[Rag80] Ragsdale, William F. 1980. fig-FORTH Installation Manual. Forth Interest Group, San Carlos, California.

[Ric87] Rich, Albert. 1987. muLISP-87 LISP Language Programming Environment. Soft Warehouse, Inc., Honolulu, Hawaii.

[Sit90] Sitaram, Dorai & Felleisen, Matthias. 1990. "Control Delimiters and their Hierarchies." Lisp and Symbolic Computation. Kluwer, Netherlands. Vol 3 No 1 pp 67-99.

[Sta87] Stallman, Richard. 1987. GNU Emacs Manual. Free Software Foundation, Cambridge, Massachusetts. prep.ai.mit.edu

[Ste90] Steele, Guy. 1990. Common LISP, The Language, 2nd edition. Digital Press, Cambridge, Massachusetts.

[Sym84] Symbolics, Inc. 1984. Symbolics 3600 Technical Summary. Symbolics, Inc., Cambridge, Massachusetts.

[Tel87] Tello, Ernest R. 1987. "Object-Oriented Programming in AI" Dr. Dobb's Journal, M Publishing Co. Redwood City, California. Apr '87, No 126.

[Tel89] Tello, Ernest R. 1989. Object-Oriented Programming for Artificial Intelligence. Addison-Wesley, Reading, Massachusetts.

[Tin86] Ting, C. H. 1986. Inside F83. Offete Enterprises, Inc. San Mateo, California.

[Tin89] Ting, C. H. 1989. System Guide to FIGForth. Offete Enterprises, Inc. San Mateo, California.

[Tin89a] Ting, C. H. 1989. F-PC 3.5 Technical Reference Manual, 2nd ed. Offete Enterprises, Inc. San Mateo, California.

[Ung91] Ungar, David & Smith, Randall B. 1991. "SELF: The Power of Simplicity." Lisp and Symbolic Computation. Kluwer, Netherlands. Vol. 4 No. 3 pp 187--205.

Glossary

After Demon A demon that executes after the main method. Useful for postprocessing operations peculiar to a particular class or object, such as type conversion of results, or maintaining the integrity of a database or table, etc.. Also useful for debugging purposes, as it may be used to log or display the results of an operation independently of the main operation itself.

Before Demon A demon that executes before the main method. Useful for preprecessing operations peculiar to a particular class or object, such as type conversion of input parameters. Also useful for debugging purposes, as it may be used to log or display the arguments to a function before the function is called.

Binding Binding is the process of forming an association between a name and an action. This may be the responsibility of the compiler, or of the executing program. With a late binding system, a name may be bound to an action at any point during the execution of a program.

Class A class is a model for an object. Many different objects may be generated that all follow the pattern of the class. Thus, a class is a sort of defining word for objects. In Dreams, a class is called a trance.

Dream A dream is an altered situation that looks pretty much like reality except that a few things are different. In a dream, the definitions of certain specified words are altered. For variables and other data objects, this alteration may be obtained by providing a local data storage area for these words. This area is private to a particular dream. For any definition, the alteration may also be a reference to a different named or unnamed word. This permits the behavior of colon definitions to be modified within the setting of a dream.

Dynamic Binding Dynamic binding is binding that occurs during the execution of a program. Which action is associated with a particular name is a function of what path the execution of the program has taken, that is, the dynamic execution of the program.

Early Binding Early binding occurs when an action is bound to a name sometime prior to the execution of that name. The earliest binding would be at compile time, when the compiler makes the association. Early binding can also occur during program execution if dynamic binding facilities are available. In this case, the binding would be early if the association between a name and an action occured in a different understanding from that in which it was used.

Essence The essence of a dream is the data structure that defines the local objects of that dream. The essence of a dream is an object that may be manipulated and given a name. Due to the dynamic scoping of dreams, the essence of a dream is insufficient to fully characterize a dream: the dream only takes on a meaning within another dream, or reality itself. The understanding of a dream is only defined while the dream is actually occuring, and is a function of the essence of the dream together with the understanding in which the dream occurs.

Imagine A dream may imagine that the definition for a word is different from reality. When this occurs, the reference that the dream makes for that word is bound to a different definition than the reference that is bound in reality.

Inheritence Inheritence permits a dream to acquire default behavior without having to explicitly specify that behavior. It is a powerful factoring technique to reduce the complexity of programs. Inheritence occurs when a dream is created, since the behavior of all its locally known objects matches that of the understanding in which the creation occurs. Inheritence also occurs when a dream is created from a class defining word, since the behavior of the class is passed on by the act of copying the class essence to make the dream's essence. Both of these kinds of inheritence occur at the birth of a dream. This is inheritence from parents. Another kind of inheritence occurs when a thought is pondered in the understanding of a dream. The understanding of a dream inherits the understanding in which the dream occurs. Any locally known words in that dream have locally understood behavior which overshadows, or overloads, the dynamically inherited behavior of those words passed on at run time, but the dynamically inherited behavoir of all words that do not have locally understood behavior in that dream is still visable and in effect. This is inherited from the environment.

Late Binding Late binding occurs when the association between a name and its action is defered until the time of that word's execution. Late binding is desirable because it permits the same message to be sent to different types of objects with different methods being invoked. This in turn permits polymorphism.

Message A message is a request for an object to perform an operation. The message is sent to the object for the object to interpret in its own understanding. In Dreams, the messages are called thoughts. Thoughts are pondered within the dynamically late bound understanding of a dream.

Method A method is an operation peculiar to an object or class. In many object oriented systems, methods must be specifically declared, but in Dreams, any word may be a method, including colon definitions, variables, and defining words.

Milieu Milieu is the essence of the current dream. It is the self-referential dream object. A thought may use milieu to make reference to the dream in which it is being pondered. If such a thought is pondered in several different dreams, milieu will always provide the essence of the current dream in which the thought is being pondered.

Object An object is the recipient of a message. In dreams, the dream is the object that receives messages. Messages are pondered in the understanding of the dream. In Dreams, messages, called thoughts, are colon definitions, and they are sent to a dream by passing the execution token of the message on the parameter stack. The dream consumes the thought, along with any parameters passed on the stack under the thought, ponders it in its own understanding, and returns any results on the parameter stack just like any other Forth word.

Overloading Overloading of methods or operators occurs when a new action is bound to a name in a dream, thereby hiding the old binding. When the dream finishes, the old binding is restored, thereby making the original meaning visible again.

Polymorphism Polymorphism refers to the ability of the same message to be sent to different objects with different results. Thus the + operator might be sent to an integer to perform integer arithmetic, or to a floating point number to perform floating point arithmetic, or to a string to perform concatenation. Dreams exhibits polymorphism implicitly due to the dynamic binding nature of method determination.

Ponder To ponder a thought is to invoke the Forth inner, or address, interpreter on a thought. A thought may be pondered in the understanding of reality, or in the understanding of a dream, or in a dream within a dream, etc. To ponder a thought is to execute its execution token within a particular understanding.

See A thought may be seen in a vision in the same manner that it may be pondered in a dream.

Reality Reality is the normal Forth programming environment, where all the usual words have all their usual meanings. This is what is usually meant when reference is made to Forth. All dreams have their origins in reality, in that they either begin in reality, or in a dream that began in reality, etc.

Regress A dream may regress by sending a thought back to be pondered within the understanding from which that dream was invoked. When that thought has been pondered, control will return to the dream that sent the thought back. This is not the same thing as sending a thought to the essence of the invoking dream, since in the first case, the active dream's understanding is unbound, the thought pondered, and then the sender's understanding is rebound, and the sender is allowed to continue the dream. In the second case, the bindings of the invoking dream are rebound as a new dream within the understanding of the sender. In this case, it is not the same dream, since the sender's understanding will be visible, whereas in the case of regression, the sender's understanding is lifted to make visible once again the invoker's understanding.

Relapse A relapse of a dream is a new instance of the essence of that dream. It is a physical copy of the data structure of a dream. This produces no new effect on alterations by reference in a dream. Where modifiable data is involved, a relapse creates a new copy of that data. The data will initially have the same value as the original, but it has a separate and distinct existence, and modifications to the original have no effect on the relaspe, nor do modifications on the relapse have any effect upon the original.

Stupor A dream about nothing. This provides a mechanism to ponder thoughts in the current understanding. It has a minimal essence, and is like any other dream in every respect.

Thought A thought is a Forth word, known by its execution token, or cfa. A thought is an object that may be manipulated and even given a name. A thought is a message that is passed to a dream. This thought is interpreted within the understanding of the dream, that is, using the dream local definitions of the words in the thought.

Understanding An understanding is the ensemble of all the currently visible Forth words and their associated definitions. In reality, the understanding is the normal Forth. In a dream state, this understanding is altered because of the altered bindings of those words whose meanings have been imagined inside the dream.

Vision A vision is an ordered set of dreams. Visions provide a cleanly packaged way to cause a thought to be pondered in a set of nested dreams. Visions provide a simple way to specify and control a network of multiple inheritence.

Advertised Interface

{ ***word1 ... wordn*** **}** (→ *cfa*)
This word compiles a colon definition from ***word1 ... wordn*** and compiles a literal holding the execution token of that colon definition. Thus an unnamed colon definition is produced. This is similar to the lambda special form of Lisp.

2VAR[***v1 ... vn*** **]** (→ *size1 cfa1 ... sizen cfan*)
This word is like **VAR[** except that it is used for **2VARIABLE**s.

AFTER (*after-cfa method-cfa ^essence* →)
Attaches an after method to the main method specified in the understanding of the dream specified by the essence pointer.

BEFORE (*before-cfa method-cfa ^essence*)
Attaches a before method to the main method specified in the understanding of the dream specified by the essence pointer.

COMA (*guzintas... thought* → *guzoutas...*)
This is the null vision. It is a vision of zero dreams. It is an object like any other vision. It may be used as a place holder in a set of multiply nested visions, or wherever an empty vision is needed.

Copy-Essence (*^old-essence* → *^new-essence*)
This word takes a pointer to the essence of a dream and makes a copy of that essence, returning a pointer to the new copy.

DEJA-VU (*guzintas... thought* → *guzoutas*)
Lifts the current understanding back to the previous vision a vision at a time. Like **REGRESS**, only works on visions instead of dreams.

DID (*pfa* →)
This word executes a colon definition when given its pfa, in much the same fashion that **EXECUTE** executes a colon definition when given its cfa. This word is used in conjunction with **REALLY** to execute the definition of a colon definition that was visable during compilation instead of during dynamic execution.

DREAM ***name*** (*NIL sizen cfan ... size1 cfa1*)
This is the defining word for a dream. It takes a **NIL**-terminated list of size/cfa pairs on the stack. The cfa's specify words to which the dream is to attach local meaning, and the size's specify how much private storage is to be allocated to the parameter field of each word.

EARLY (*word-cfa essence* → *word-pfa*)
Compiles the compile-time binding of word-cfa in the understanding of essence. This permits early binding to the understanding of a dream other than the one that is executing while the compilation is occuring.

ESSENCE *dream* (→ *^essence*)
This word is a macro to extract the essence of a dream. It reads the next source stream input token, and assuming that this is the name of a dream, it returns a pointer to the data structure for that dream.

FANTASY (*vision* → *dream*)
Converts a vision into a dream that has identical bindings, except that variables will have new pfa's.

IMAGINE (*new-cfa old-cfa ^essence*)
This word is used to modify a dream's understanding of a word local to that dream. The word whose name is given by old-cfa will cause the action given by new-cfa when pondered in the dream given by essence.

Make-Essence (*NIL sizen cfan ... size1 cfa1* → *^essence*)
This word takes a **NIL**-terminated list of objects to be made local to a dream, together with each of their types, represented by the object's size, and allocates storage for and initializes a data structure for the essence of a dream. A pointer to this data structure is returned on the stack.

make-vision (*NIL ^essence1 ... ^essencen* → *vision*)
This word makes a vision from a **NIL**-terminated list of dream essence pointers, and returns a pointer to it on the stack.

MILIEU (→ *^essence*)
This word returns the essence of the currently active dream. It is the self-referential dream.

NIL (→ *null-pointer/false*)
This word returns a pointer to nowhere, which is also interpreted as a false logical value. Thus in ANS-Forth, it is zero.

PONDER (*guzintas... thought ^essence* → *guzoutas*)
This word causes a thought to be pondered in the understanding of a dream. It takes a pointer to the essence of the dream, a thought, and any input parameters to that thought. It returns the result of applying that thought in the understanding of the dream to the input parameters.

REALITY (*guzintas... thought* → *guzoutas*)
This word ponders a thought in reality by regressing repeatedly until the true understanding of reality is reached.

REALLY *word* (→ *pfa*)
This word compiles a literal containing the pfa of the word whose name follows the word **REALLY** in the input stream. It is used to force early binding to words that would otherwise have a dynamically bound local definition.

REF[*name1 ... namen* **]** (→ *size1 cfa1 ... sizen cfan*)
This word is a macro like **VAR [** and **2VAR [** only it is used for reference bindings. These are used for colon definitions and other words that do not have a modifiable data storage area associated with them.

REGRESS (*guzintas... thought* → *guzoutas*)
This word is used to cause a thought to be pondered in the understanding in which the current dream was invoked. It causes the understanding of the current dream to be lifted, or undone,

thereby making the understanding in which that dream was invoked again visible. After the thought has been pondered, the current dream's understanding is restored.

RELAPSE *new-dream* (*^essence* →)
This defining word creates a new named copy of a dream whose essence is passed to it. It reads the next source stream input token and creates a dictionary entry for that name as a dream which is a copy of the original dream.

SEE (*guzintas... thought ^vision* → *guzoutas...*)
This word causes a thought to be seen in a vision the same way that **PONDER** causes a thought to be pondered in a dream. It may also be used to ponder a thought in a dream, since a dream is an atomic vision; or equivalently, visions are molecules formed of dreams.

STUPOR (*guzintas... thought* → *guzoutas...*)
This word is a dream with no local meanings. It may be used to cause a thought to be pondered in the current understanding. Its essence may be manipulated exactly like any other dream. It is the milieu of reality.

THOUGHT (*cfa* →)
This is a defining word that is used to give a name to a thought. When that name is later used, it returns the associated thought as an execution token, or cfa.

TRANCE *name* (*NIL sizen cfan ... size1 cfa1* →)
This is a meta-defining word. It is used to create a word that creates dreams with similar essences. **TRANCE** reads the next source stream input token and creates a word of that name which is itself a defining word. When the offspring word is used, it reads a token from the input stream and creates a dream of that name with the local objects specified to the trance. In conventional object oriented parlance, **TRANCE** is a class defining word.

VAR[*var1 ... varn* **]** (→ *size1 cfa1 ... sizen cfan*)
This word is a macro to ease coding and improve readability. It is used to build a list of variables for **Make-Essence**. The syntax of its use is as follows: **VAR[** *var1 ... varn* **]**. See the example session for an example of its use.

VISION *name* (*NIL ^essence1 ... ^essencen* →)
This is the defining word for a vision. The essences of the dreams comprising the vision are passed on the stack, with the dominant dream pushed first.

VISION[*dream1 ... dreamn* **]** *name* (→)
This is a macro for declaring visions at compile time. It takes a list of dream names from the source stream and builds a vision with the specified name.

[EARLY] *word dream* (→)
Compiles the compile-time early binding of word in the understanding of dream.

Listing 1

This listing is a log of an interactive demonstration run. The indented lines are inputs to the Forth system, and the non-indented lines are the output from the Forth system.

```
   VARIABLE X    1 X !              \ make some variables...
   VARIABLE Y    2 Y !
   VARIABLE Z    3 Z !
   : ? @ . ; ( variable -- )        \ print the value of a variable
   : DOT ( n -- )                   \ another way to print a number
     0 <# # # # # #>                \ with leading zeros
     TYPE CR ;                      \ and a carriage return
   NIL VAR[ X Y ] DREAM snooze       \ a dream about a couple of variables
   ESSENCE snooze RELAPSE nap        \ a different version of the same dream
   { X ? Y ? Z ? } THOUGHT hmm       \ a thought about 3 variables
   hmm STUPOR                        \ ponder it in reality
1 2 3
   hmm snooze                        \ and again in a dream
1 2 3
   100 X !                           \ change the variables
   200 Y !
   300 Z !
   hmm STUPOR                        \ ponder it again in reality
100 200 300
   hmm snooze                        \ and again in the dream
 1 2 300
   { 10 X ! 20 Y ! 30 Z ! } snooze \ make a change in the dream
   hmm snooze                        \ what was the effect?
10 20 30
   hmm STUPOR                        \ what was the effect on reality?
100 200 30
   hmm nap                           \ what is the other dream thinking?
1 2 30
   NIL REF[ . ] TRANCE FUE           \ a class of dreams about a colon definition
   FUE FOO                           \ a dream of that class
   FUE BAR                           \ another dream of the same class
   hmm FOO                           \ try it out
100 200 30
   hmm BAR                           \ try that one too
100 200 30
   ' DOT ' . ESSENCE FOO IMAGINE     \ imagine that . behaves differently
   hmm FOO                           \ exhibit the new behavior
0100
0200
0030
   hmm BAR                           \ show that BAR was unaffected
100 200 30
   { hmm BAR ]  FOO                  \ a dream within a dream
100 200 30
   { hmm FOO ] snooze               \ same dream, but within a different dream
0010
0020
0030
   { { hmm BAR ] FOO ] snooze       \ this can go as far as you like!
10 20 30
   { hmm nap ] FOO                  \ showing FOO's . still holds in nap
0001
0002
0030
   { hmm REGRESS ] FOO               \ regress back from foo to reality
100 200 30
   { { hmm nap } REGRESS } FOO       \ more advanced navigation of the dream world
1 2 30
   [ X ? ] snooze                    \ show snooze's understanding of the variable
10
   { REALLY X ? ] snooze             \ show early binding to a variable
100
   { X @ . } FOO                     \ show FOO's understanding of .
0100
```

```
   { X @ REALLY . DID } FOO       \ show early binding to a colon definition
100
   VISION[ BAR FOO snooze ] BAZ   \ make a vision to show how that works
   hmm BAZ                        \ should be same as:
10 20 30                          \ { { hmm BAR } FOO } snooze
   VISION[ FOO BAR snooze ] BOK   \ should print with leading zero
   hmm BOK                        \ try it and see
0010
0020
0030
   hmm ESSENCE snooze SEE         \ test of atomic dream passed to SEE
10 20 30
   VISION[ FOO BAZ ] BACH         \ make a nested dream
   hmm BACH                       \ test of a nested dream
0010
0020
0030
   VISION[ FOO COMA BACH ] HANDEL \ doubly nested dream with null in middle
   hmm HANDEL                     \ try that one on for size!
0010
0020
0030
   VISION[ BAR HANDEL ] MOZART    \ exception link to override FOO
   hmm MOZART                     \ show that it works
10 20 30
   { hmm REALITY } HANDEL         \ test reality word
100 200 30
```

Listing 2 *DREAMS.4TH*

```
   (
                                  DREAMS
                        An Object Oriented System
                          For LMI UR/Forth 1.03
                         Written By:  R. J. Brown
                   Copyright 1989 Elijah Laboratories Inc.
                        All Rights Reserved Worldwide
                       This code may be freely copied and
                       distributed under the terms of the
                       Gnu Public License.  [See gnu.txt]
                       "...and the dream is certain, and
                        the interpretation thereof sure."
                                             Daniel 2:45

   The dreams system arose out of an experimental port of a Flavors and a dynamic
closures package from Lisp to Forth.
   )
\ State which prerequisite source files must be present.
CONSULT ANS                        \ X3/J14 BASIS6 compatibility for LMI UR/Forth.
CONSULT MACROS                     \ Eli Lab's macro defining words.
CONSULT EDO                        \ George Hawkins' structured data types.
CONSULT STACKS                     \ Stack defining and manipulating words.
\ Define the stacks to hold old bindings and active closures.
100 Stack ABStk                    \ active bindings during dreams
 25 Stack AEStk                    \ active essences during dreams
 25 Stack UEStk                    \ unbound essences during regressions
 10 Stack AVStk                       \ active visions stack
```

```
\ Define data types used in the structure of an essence of a dream.
cell  DEF pointer                  \ an address of something else
\ Define the data structure for a dream's essence.
S{ cell :: dream-size              \ the size of this dream
   cell ::  #-of-bindings          \ the number of local objects
   \ a dream has one of these slots for each local binding
   S{ pointer :: pfa-pointer       \ points at a pfa we are closed over
      pointer :: pfa-contents      \ our local value for that pfa
      cell    :: local-type  }S    \ the type of this local
      DUP DEF local-binding        \ the type of a slot
      [*]     local-binding[]      \ the slot index operator
   ::  local-bindings              \ the name the vector of slots
}S local-binding - DEF dream-header  \ this is called a dream-header
( There is one slot in the local-bindings vector for each locally bound object.
Following this, a region of dictionary is ALLOTed to hold the BODYs of each of the
locally bound objects.  An ALLOTment is made for each object equal to the size of
that object, which is also the object's type.  Reference type bindings have a size
of zero.)
\ LMI UR/Forth memory model dependent words.
: ^pfa ( cfa -- ^pfa ) BYTE+ ;       \ Convert a cfa to a ptr to the pfa.
: pfa@ ( ^pfa -- pfa ) CS0 SWAP @L ;  \ Fetch a pfa from the code segment.
: pfa! ( pfa ^pfa -- ) CS0 SWAP !L ;  \ Store a pfa into the code segment.
\ Instantiate the essence of a dream and return a pointer to it on the stack.
: Make-Essence ( NIL size-n cfa-n ... size-1 cfa-1 -- ^essence )
   HERE >R                        \ save pointer to instantiation
   dream-header ALLOT             \ allocate the header
   BEGIN ?DUP WHILE               \ for each locally bound object...
     HERE >R                      \ remember start of slot
     local-binding ALLOT          \ allocate a local binding slot
     ^pfa R@ pfa-pointer !        \ store pointer to pfa
     R> local-type ! REPEAT       \ store the object's length
   HERE R@ local-bindings -       \ compute size of local binding vector
   local-binding /                \ compute number of local bindings
   R@  $-of-bindings !            \ save it for dynamic binding routines
   HERE                           \ point to start of local data area
   R@ local-bindings              \ point to the local-bindings vector
   ?DO HERE I pfa-contents  !     \ set pointer to local data slot
     I local-type @ ALLOT         \ reserve space for it
     I pfa-pointer @ pfa@         \ point to original data
     I pfa-contents @             \ point to new data slot
     I local-type @ MOVE          \ get length & copy data to new slot
     I local-type @ 0=            \ is it a reference binding?
     IF I pfa-pointer @ pfa@      \ yes, inherit old pfa
        I pfa-contents ! THEN     \ instead of copy of data
     local-binding +LOOP          \ repeat for each local object
   HERE R@ -                      \ compute overall size of this essence
   R@ dream-size !                \ store for future RELAPSE calls
   R> ;                           \  return pointer to this essence
\ Copy an essence to produce a new essence with the same initial bindings.
: Copy-Essence (  ^old-essence --  ^new-essence )
   HERE                           \ destination address
   2DUP OVER dream-size @         \ length to copy
   DUP ALLOT                      \ allocate space
   MOVE                           \ make the copy
   SWAP OVER -                    \ compute ptr adjustment
   OVER  #-of-bindings @ 0        \ for each local binding
   ?DO OVER local-bindings        \ point to...
     I SWAP local-binding[]       \     ...its slot
     DUP local-type @             \ locally instantiated?
     IF pfa-contents DUP @            \ yes, get old binding
        2 PICK - SWAP !               \ adjust to new binding
     ELSE DROP THEN LOOP DROP ;       \ loop till done
\ Establish new bindings for local objects.
: new-bindings ( ^essence -- )
   DUP AEStk Push                     \ stack dream occurrence
   DUP local-bindings SWAP            \ point to bindings vector
   #-of-bindings @ 0                  \ for each local binding
   ?DO I OVER local-binding[]         \ point to its slot
     DUP pfa-pointer @                \ point to its pfa
     DUP pfa@ ABStk Push              \ save old binding
     SWAP pfa-contents @              \ get new binding
     SWAP pfa!                        \ establish new binding
     LOOP DROP ;                      \ clean up   exit
\ Re-establish stacked old bindings for local objects.
: old-bindings ( -- )
   AEStk Pop                        \ point to most recent dream
   DUP local-bindings SWAP          \ point to bindings vector
   #-of-bindings @ ?DUP 0=          \ are there any bindings?
   IF DROP EXIT THEN                \ no, do nothing and exit
   1- 0 SWAP                        \ yes, for each local binding
   DO I OVER local-binding[]        \ point at its slot
     pfa-pointer @                  \ point at pfa
     ABStk Pop                      \ get stacked old binding
     SWAP pfa!                      \ restore old binding
     -1 +LOOP DROP ;                \ clean up   exit
\ Cause a dream to ponder a thought.
: PONDER ( guzintas... thought-cfa ^essence -- guzoutas... )
   new-bindings                     \ bind to local objects
   EXECUTE                          \ do your thing!
   old-bindings ;                   \ unbind from local objects
\ Declarations of lists of objects local to a dream.
: VAR[ { cell ' } [COMPILE] IRP[ ;      \ declare variables
: 2VAR[ { 2 CELLS ' ] [COMPILE] IRP[ ; \ declare 2variables
: REF[ { NIL ' }  [COMPILE] IRP[ ;      \ declare reference bindings
\ Defining word for a dream.
: DREAM ( NIL size-n cfa-n ... size-1 cfa-1 -- )  \ name \
   Make-Essence CREATE ,           \ give it a name
   DOES @ PONDER ;                 \ and a behavior
\ A dream about nothing provides a way to ponder thoughts in the here and now.
NIL DREAM STUPOR             \ ponders thoughts in the current understanding
\ Transform a dream's name to its essence, or data structure address.
: ESSENCE ' >BODY @ STATE @ IF [COMPILE] LITERAL THEN ; IMMEDIATE
\ Defining word for a copy of a dream.
: RELAPSE ( ^essence -- ) \ new-dream \
   \ Syntax:   ESSENCE old-dream RELAPSE new-dream
   Copy-Essence CREATE , DOES> @ PONDER ;
\ Defining word for a class of dreams.
: TRANCE ( NIL size-n cfa-n ... size-1 cfa-1 -- ) \ name \
   Make-Essence CREATE ,                 \ make the prototype
   DOES> @ RELAPSE ;                     \ replicate the prototype
\ Reference to the essence of the current dream.
: MILIEU ( -- ^essence )
   AEStk Empty?                          \ is there currently any active dream?
   IF ESSENCE STUPOR ELSE                \ no, return empty dream's essence
     AEStk Top THEN ;                    \ yes, return current dream's essence
\ Defining word for a named thought.
```

```
: THOUGHT CONSTANT ;                        \ Syntax:   { blah blah blah } THOUGHT name
\ Find a word's local-binding slot in an essence.
: Find-Binding ( pfa ^essence -- ^slot )
   DUP local-bindings >R                    \ save pointer to vector
   #-of-bindings @ ?DUP 0=                  \ is number of slots zero?
   IF R> 2DROP NIL EXIT THEN                \ yes, give up search!
   BEGIN DUP 1- R@ local-binding[]          \ no, point to slot
     DUP pfa-pointer @ 3 PICK =             \ is this right slot?
     IF -ROT 2DROP R> DROP EXIT             \ yes, return its pointer!
     ELSE DROP THEN                         \ no, keep looking
     1- ?DUP 0= UNTIL                       \ update index
   DROP NIL ;                               \ search failed, return NIL!
\ Alter the understanding of an object local to a dream.
: IMAGINE ( new-cfa old-cfa ^essence -- )
   SWAP ^pfa SWAP                           \ point to old pfa pointer
   Find-Binding ?DUP                        \ find its binding in the essence
   IF DUP local-type @                      \ make sure its a reference binding
     IF DROP EXIT THEN                      \ if not reference, don't bind it
     SWAP ^pfa pfa@ SWAP                    \ point to new pfa
     pfa-contents !                         \ replace old pfa with new pfa
   ELSE DROP THEN ;                         \ if not found, do nothing
\ Regress back to an earlier dream state.
: REGRESS ( guzintas... thought -- guzoutas... )
   AEStk Empty?                             \ are we already in reality?
   IF EXECUTE                               \ yes, can't regress further
   ELSE AEStk Top UEStk Push                \ no, remember where we are
     old-bindings                           \ go back a level
     EXECUTE                                \ ponder the thought there
     UEStk Pop new-bindings THEN ;          \ return to where we came from
\ Regress all the way back to Reality.
: REALITY ( guzintas... thought -- guzoutas... )
   BEGIN AEStk Empty? NOT WHILE             \ till there's no bindings left
     AEStk Top UEStk Push                   \ remember what we undid
     old-bindings REPEAT                    \ un-do a binding
   EXECUTE                                  \ think the thought
   BEGIN UEStk Empty? NOT WHILE             \ till they're all re-bound
     UEStk Pop                              \ get a binding
     new-bindings REPEAT ;                  \ re-bind it.
\ Early binding support: compile time pfa value.
: REALLY ( -- pfa )  \ name \
   ' >BODY [COMPILE] LITERAL ; IMMEDIATE        \ pfa of name in reality
\ Execute early bound colon definition.
: DID ( pfa -- )  \ syntax:  REALLY word DID \
   R> DROP >R ;                             \ execute body and return
\ Build data structure for a vision, which is a set of dreams.
: make-vision ( NIL ^essence-1 ... ^essence-n -- ^vision )
   HERE >R                                    \ remember where vision starts
   NIL ,                                      \ backwards terminator
   BEGIN ?DUP WHILE                           \ for each dream in the vision
        , REPEAT                              \ remember its essence
   NIL ,                                      \ forward terminator
   R> ;                                       \ return pointer to vision
\ Establish the understanding of a vision.
: bind-vision ( ^vision -- )
   DUP @ IF DUP new-bindings                  \ handle atomic dream case
     ELSE BEGIN CELL+ DUP @ ?DUP WHILE        \ for all dreams in vision
        DUP @ IF new-bindings                 \ handle dream in this slot
          ELSE RECURSE THEN                   \ handle nested vision
```

```
      REPEAT THEN AVStk Push ;          \ remember tail for unbinding
\ Disestablish the understanding of a vision.
: unbind-vision ( -- )
   AVStk Pop                            \ point to the vision's tail
   DUP @ IF old-bindings                \ handle atomic dream case
     ELSE BEGIN CELL- DUP @ WHILE       \ for all dreams in vision
        DUP @ @ IF old-bindings         \ handle dream in this slot
           ELSE RECURSE THEN            \ handle nested vision
        REPEAT THEN DROP ;              \ clean up before exit
\ See a thought in a vision.
: SEE ( guzintas... thought ^vision -- guzoutas... )
   bind-vision                          \ establish the understanding of the vision
   EXECUTE                              \ ponder the thought therein
   unbind-vision ;                      \ remove the understanding of the vision
\ Defining word for a vision, arguments are on the stack.
: VISION ( NIL ^essence-1 ... ^essence-n -- )
   make-vision CREATE ,
   DOES> @ SEE ;
\ A vision about nothing makes a useful place-holder in another vision.
NIL VISION COMA                      \ analogous to STUPOR, the dream about nothing
( Note that the essence of a vision may be extracted just like the essence of a
dream. The use of the word ESSENCE is exactly the same in both cases.)
\ Defining word for a vision, arguments are names in the source stream.
: VISION[ ( -- ) \ Syntax:  VISION[ dream-1 ... dream-n ] name
   NIL ['] ESSENCE [COMPILE] IRP[ VISION ;
\ Words to permit early binding to the understanding of another dream.
: EARLY ( word-cfa ^essence -- word-pfa )
   ['] >BODY SWAP PONDER ;
: [EARLY] ( -- ) \ word-name dream-name \
   ' [COMPILE] ESSENCE EARLY [LITERAL] ; IMMEDIATE
\ Words to plant before and after demons into other words.
: BEFORE ( before-cfa method-cfa ^essence -- )
   2DUP EARLY                              \ get old method
   HERE >R SWAP >R                         \ save cfa & essence ptr
   ROT , [LITERAL] COMPILE DID COMPILE EXIT \ compile new method
   R> R> -ROT IMAGINE ;                    \ replace old method with it
: AFTER ( after-cfa method-cfa ^essence -- )
   2DUP EARLY                              \ get old method
   HERE >R SWAP >R                         \ save cfa & essence ptr
   ROT SWAP [LITERAL] COMPILE DID , COMPILE EXIT \ compile new method
   R> R> SWAP IMAGINE ;                    \ replace old method
```

Listing 3 *ANS.4TH*

```
(
              BASIS6 Compatibility Suite for LMI UR/Forth 1.03
                          Written by: R. J. Brown
                     Copyright 1989 Elijah Laboratories Inc.
          See the ANS X3/J14-TC BASIS6 document for stack effects and
                    other documentation regarding these words.
  )
: 2>R >R -ROT >R >R >R ;
: 2R> R> R> R> ROT >R ;
: [ASCII] [COMPILE] ASCII ; IMMEDIATE
\ : ASCII [COMPILE] [ASCII] ;
\ : ASSEMBLER ASM ;
```

```
: BYTE+ 1+ ;
: BYTES ;
: CELL+ WSIZE + ;
: CELLS WSIZE * ;
1 CELLS CONSTANT CELL
\ : D! 2! ;
: D>S DROP ;
\ : D@ 2@ ;
( Ray Duncan says he has a workable definition for this word... )
\ : EVALUATE ." EVALUATE is not yet implemented! " ABORT ; IMMEDIATE
: FOR COMPILE 0 [COMPILE] DO ; IMMEDIATE \ This is fudged! A real FOR-NEXT
: NEXT [COMPILE] LOOP ;                  \ loop doesn't alter the R-stack!
: MOVE ?DUP IF >R 2DUP R@ -  IF R> CMOVE>
    ELSE R> CMOVE THEN
  ELSE 2DROP THEN ;
: OCTAL 8 BASE ! ;
: UNDO R> R> R> 3DROP ;
\ The following words are not ANS X3/J14, but are close relatives.
\ They are included for the sake of convenience here.
1 CELLS CONSTANT cell         \ a single forth virtual machine cell
: CELL- cell - ;              \ back up a pointer by one cell
```

Listing 4 *EDO.4TH*

```
(
                    Extended Data Objects for Forth.
                       Written by George Hawkins
                   Moderator of the "Fifth" Conference
                      On the East Coast Forth Board
             Modified by R. J. Brown, Elijah Laboratories Inc.
             Modified to be compatible with ANS X3/J14-TC BASIS6,
      and also to permit more flexibility in structure declarations,
                   especially with regard to vectors.
)
: DEF ( type -- ) \ C \ Define a scalar type.
  \ <type> \
  ( -- type )                  \ R
  CONSTANT ;
: S{ ( -- offset )             \ Initiate a structure definition.
  0 ;
: :: ( old-offset type -- new-offset ) \ C \ Define a structure component.
  \ <component> \
  ( -- new-offset ) \ R
  CREATE OVER , +
  DOES> @ + ;
:  )S ( offset -- type )       \ End a structure definition.
  ;
: [*] ( object-definition -- ) \ C \ Define a vector operator.
  \ <vector-operator> \
  ( index base-addr -- element-addr ) \ R
  DUP CREATE ,
  DOES> @ ROT * + ;
: DEF[] ( element-type  elements -- ) \ C \ Define a vector.
  \ <vector-type> \
  ( -- vector-type ) \ R
  * CONSTANT ;
```

Listing 5 *MACROS.4TH*

```
(
                    MACRO Support for LMI UR/FORTH
                          Copyright (c) 1988
                       Elijah Laboratories Inc.
                             Written by:
                             R. J. Brown
                       Elijah Laboratories Inc.
                         201 West High Street
                            P. O. Box 833
                           Warsaw KY 41095
                           1 606 567-4613

  This file defines a words useful for the writing of defining and compiling words.
Especially noteworthy is the EVAL word that allows processing of one token in the
input stream by the outer interpreter, and then returns control to the word that in-
voked EVAL.
  )

CONSULT UTIL                                              \ prerequisite modules

( This word is used by the override words for bases and vocabularies. It takes the
address of a variable and a new value for that variable, and returns the old value
of that variable and its address so that the old value may be restored with a sim-
ple ! operation. )

: XCHG DUP >R @ SWAP R@ ! R> ;                           ( new addr -- old addr )

( These words will evaluate one word from a text string, and one word from the
input stream. They are useful for overriding things like the BASE or the VOCABULARY
that is normally in effect, and then restoring it after that one word has been eval-
uated. )

: eval  FIND CASE                        ( str -- ; evaluate the word in str )
   0  OF NUMBER? 0= ABORT" is undefined! " DROP                    ( number )
      STATE @ IF [COMPILE] LITERAL THEN ENDOF
  -1  OF STATE @ IF , ELSE EXECUTE THEN ENDOF                        ( word )
   1  OF EXECUTE ENDOF ENDCASE ;                           ( immediate word )

: EVAL BL WORD eval ;                        ( -- ; read and evaluate a word )

( These words allow the current base to be overridden for the execution/interpreta-
tion/compilation of the next word from the input stream. They restore the original
base when the overridden word is finished executing.  )

: base'                   ( n -- ; causes next word to operate in base n )
   BASE XCHG >R >R EVAL R> R> ! ;

( Compact forms for the most popular bases... )

: X' 16 base' ; IMMEDIATE                            ( force hexadecimal )
: D' 10 base' ; IMMEDIATE                                ( force decimal )
: O'  8 base' ; IMMEDIATE                                  ( force octal )
: B'  2 base' ; IMMEDIATE                                 ( force binary )

( These words work in a fashion analogous to the base overriding words, only they
override the vocabulary instead of the base, restoring it after the next word has
been executed. )

: v'                                             ( " <vocab>" v' <word> )
   CONTEXT @ >R                                        ( save orig vocab )
   eval                                             ( execute temp vocab )
   EVAL                                ( read and execute overridden word )
   R> CONTEXT ! VOCORDR ;                           ( restore orig vocab )

: V'     BL WORD v' ; IMMEDIATE                       ( V' <vocab> <word> )

( This special version of CREATE will act the same way CREATE does unless the name
read from the input stream is *not-used* in which case QREATE will not create a dic-
tionary header and will exit not only itself, but also the word that called it. The
value returned by *not-used*? is true if QREATE found the special "*not-used*"
token in the input stream. Embeded comments are handled in the expected way, and
not treated as names to be created.)

VARIABLE    ?not-used?             ( *not-used* flag, T if not CREATEd )
```

```
: *not-used*? ?not-used? @ ;              ( predicate is T if no CREATE )
: _qr                                      ( factored helper for QREATE )
  R> R> DROP >IN @ >R >R NIL ;                ( update IN for comments )
: QREATE                      ( -- ; Qreate  ame ... a queer CREATE !!! )
  >IN @ >R                            ( remember place in input stream )
  ?not-used? NIL!                        ( assume we will do a CREATE )
  BEGIN BL WORD DUP                                  ( read next token )
    COUNT " *not-used*" COUNT STRCMP                   ( special case? )
    0= IF R> 2DROP                              ( yes, clean up stack, )
    ?not-used? T! 2EXIT THEN               ( set flag,   double return )
    FIND IF CASE                         ( allow embedded comments... )
      ['] ( OF [COMPILE] ( _qr ENDOF                     ( parenthesis )
      ['] \ OF [COMPILE] \ _qr ENDOF                      ( back-slash )
      T SWAP                                     ( none of the above... )
    ENDCASE ELSE DROP T THEN                        ( token not found at all )
  UNTIL                            ( keep looking for non-comment tokens )
  R> >IN !                                     ( restore input stream )
  CREATE ;                                        ( do a normal CREATE )
( The following words make use of the QREATE word to implement conditionally gener-
ated constants and variables. These are particularly useful when macro-type words
generate a family of constants or variables, and certain of the members of these
families are not really used. It is nice to have *not-used* as a place holder for
the vacant slots, without generating unneeded dictionary headers and a host of 'is
re-defined' messages. )
: QONSTANT QREATE , DOES> @ ;                        ( conditional constant )
: 2QONSTANT QREATE , , DOES> 2@ ;               ( double precision const. )
: QVARIABLE QREATE 2 ALLOT DOES> ;                   ( conditional variable )
\ The Forth version of the indefinate repeat macro.
: IRP[ ( cfa --\ { words to repeat }  IRP[ tkn-1 ... tkn-n ] )
  >R                                                           \ save cfa
  BEGIN                                           \ for the following tokens
    >IN @ >R                                          \ save input pointer
    BL WORD COUNT                                    \ read the next token
    " ]" COUNT STRCMP WHILE                       \ until we encounter a ']'
    R> >IN !                                      \ back up to token again
    R@ EXECUTE                                 \ apply the cfa to the token
  REPEAT                                                   \ loop till done
  R> DROP                                        \ trash saved input pointer
  R> DROP                                                 \ trash saved cfa
: IMMEDIATE                                        \ this is a read macro word
\ Give names to bit masks in order read.
: BITS[ ( --   BITS[ bit0 bit1 bit2 ... bitn ] )
  1                                                   \ mask for first bit
  { DUP QONSTANT 2* }                                              \ do this
  [COMPILE] IRP[                                        \ for each bit name
  DROP ;                                              \ trash leftover mask
\ Give indices to symbols, starting with ival   returning oval.
: ENUM[ ( --   ival ENUM[ tag1 tag2 ... tagn ] -- oval )
  { DUP QONSTANT 1+ }                                              \ do this
  [COMPILE] IRP[ ;                                            \ for each tag
\ A block compile word.
: COMPILE[ ( -- \ COMPILE[ word1 ... wordn ] )
  {                                                          \ for each token
    COMPILE COMPILE                                   \ compile the value of
    EVAL                                                  \ its evaluation
  }                                                 \ this is the action to repeat
  [COMPILE] IRP[                                  \ until a ']' is encountered
; IMMEDIATE                                        \ this is a read macro word
```

```
( This word is a macro to declare several VARIABLEs at a time. It is used as fol-
lows: VARS[ var1 var2 ... varn ] )
: VARS[                                         \ multiple variable declaration
   ['] VARIABLE                                         \ perform VARIABLE
   [COMPILE] IRP[ ;                                \ for each token in the list
( This word is a macro to declare a list of tokens to be forward references. These
words must later be resolved with the R: word.
 !!! LMI UR/FORTH Only !!! )
: FWD[                                        \ declare many forward references
   ['] F:                                                   \ perform F:
   [COMPILE] IRP[ ;                                \ for each token in the list
( These words are used to create a "stub" word that just displays its name when it
is executed. These words are useful when doing a top-down implementation with test-
ing before all words are coded. )
VARIABLE here                  \ variable needed to thwart compiler security
: STUB HERE here ! :                                      \ stub off a word
   here @ BODY> >NAME [COMPILE] LITERAL                  \ make nfa literal
   COMPILE CR COMPILE .NAME [COMPILE] ; ;               \ code to show name
: STUB[ ['] STUB [COMPILE] IRP[ ;                      \ stub a list of words
( This word consults a list of files )
: CONSULT[ ( -- \ CONSULT[ f1 ... fn ] ; consult listed files )
   ['] CONSULT [COMPILE] IRP[ ;
```

Listing 6 *STACKS.4TH*

```
\ Defining word and methods for stacks.
CONSULT ANS                 \ ANS Forth X3J14 BASIS6 compatibility.
: Stack CREATE ( size -- )  \ stack \
   HERE CELL+ , CELLS ALLOT
   DOES> ;                    ( --  sp )
: Push ( w stack -- )       \ push a word onto a stack
   DUP >R @                 \ fetch sp
   cell +                   \ pre-increment sp
   DUP R> !                 \ save new sp
   ! ;                      \ store word in stack
: Pop ( stack -- w )            \ pop a word from a stack
   DUP >R @                     \ fetch sp
   DUP @                        \ fetch word from stack
   SWAP cell -                  \ post-decrement sp
   R> ! ;                       \ save new sp
: Top ( stack -- w )            \ fetch top of stack
   @ @                          \ fetch the word the pointer points to
: Empty? ( stack -- flag )      \ test for an empty stack
   DUP @ = ;                    \ its empty if pointer points to its self
```

Listing 7 *UTIL.4TH*

```
(
                          Utility words
                       Copyright (c) 1988
                    Elijah Laboratories Inc.
                          Written by:
                          R. J. Brown
                    Elijah Laboratories Inc.
                      201 West High Street
                          P. O. Box 833
                        Warsaw KY 41095
                         1 606 567-4613
                      rj brown @ ecfb / lmi
```

```
   This file defines various useful utility words. It is basically a catch-all repos-
itory for miscelaneous widgets.
   The material contained in this file is Copyright [c] 1988 Elijah Laboratories
Inc. All rights reserved world wide.
   Permission is hereby granted to reproduce this document in whole or in part pro-
vided that such reproductions refer to the fact that the copied material is subject
to copyright by Elijah Laboratories, Inc. No changes or modifications may be made
to the copied material unless it is clearly indicated that such changes were not in-
corporated in the original copyrighted work.
   )
( Machine independent word size tools. )
WSIZE CONSTANT 1w                  ( --  bytes/word ; machine word size )
    1 CONSTANT 1b                  ( --  bytes/byte == 1 always in F83! )
   1w CONSTANT 1W  1b CONSTANT 1B           ( allow upper or lower case )
: w+ 1w +  ; : W+  w+  ;                ( n -- n+1w ; add 1 word offset )
: w- 1w -  ; : W-  w-  ;           ( n -- n-1w ; subtract 1 word offset )
: w* 1w *  ; : W*  w*  ;                  ( n -- n*1w ; n words offset )
: w*+ w* + ; : W*+ w*+ ;           ( k n -- k+n*1w ; add n words offset )
: w/ 1w /  ; : W/  w/  ;                ( nb -- nw ;  bytes to   words )
1w 2* CONSTANT 1d             \ no upper case equiv because of name clash
: d+ 1d + ;                          \ with Forth-83 standard D-words.
: d- 1d - ;
: d* 1d * ;
: d*+ d* + ;
: d/ 1d / ;
EXISTS? FPSIZE .IF FPSIZE CONSTANT 1f  1f CONSTANT 1F
: f+ 1f +  ; : F+  f+  ; : f-  1f - ; : F-  f-  ;
: f* 1f *  ; : F*  f*  ; : f*+ f* + ; : F*+ f*+ ;
: f/ 1f /  ; : F/  f/  ;
: IFIX FIX DROP ; ( float -- int ) .THEN
\ Determine the implementation dependant pointer size.
\ HERE                                 \ dictionary position before ptr
\    NULPTR PTR P                                \ allocate a pointer
\ HERE                               \ dictionary position after ptr
\    FORGET P                                    \ get rid of pointer
\ SWAP - CONSTANT 1p               \ compute size and give it a name
   1w CONSTANT 1p                             \ for UR/Forth-386 !!!
\ Define pointer word size tools.
: p+    1p + ;
: p-    1p - ;
: p*    1p * ;
: p*+   p* + ;
: p/    1p / ;
( Words for handling segmented address space transparently. )
\ 1w 4 = .IF
\ : >S:O     ADDR>S&O ;
\ :  S:O>    S&O>ADDR ;
\ .ELSE
   : >S:O   ( Do nothing! )  ;  IMMEDIATE
   : S:O>   ( Do nothing! )  ;  IMMEDIATE
\ .THEN
( The above is for LMI Forths. Do whatever you have to here for your own favorite
brand of the language. )
```

```
( Convenient words to have around. )
       0 CONSTANT NIL                   ( -- false ; logical False value )
 NIL NOT CONSTANT T                       ( -- true ; logical True value )
: T!    T SWAP ! ;                                     ( v -- ; sets to T )
: NIL!  NIL SWAP ! ;                                 ( v -- ; sets to NIL )
: FLIP  DUP @ 0= SWAP ! ;                         ( v -- ; reverses truth )
: SETQ ' SWAP ! ;                                 ( vec --   vec SETQ word )
: ++    1 SWAP +! ;                          ( addr -- ; increments word )
: --   -1 SWAP +! ;                          ( addr -- ; decrements word )
: R++ R> 1+ >R ;                         ( -- ; increment top of R-stack )
: R-- R> 1- >R ;                         ( -- ; decrement top of R-stack )
: SWOOP SWAP DUP ;                    ( x y -- y x x ; for combining tests )
: ... ; IMMEDIATE                  ( elipsis "noise word" for stubs, etc. )
: 3DUP   2 PICK 2 PICK 2 PICK ;              ( copy top 3 stack elements )
: ?IF COMPILE ?DUP [COMPILE] IF ; IMMEDIATE
: NDUP BEGIN DUP WHILE 1-                ( ... n -- ... ... ; DUP n items )
  OVER SWAP REPEAT DROP ;
: NDROP 0 ?DO DROP LOOP ;                        ( ... n -- ; DROPs n items )
: NSWAP            ( a..b c..d n -- c..d a..b ; SWAPs n word elements )
  DUP 2* 1- SWAP
  0 ?DO
    DUP >R ROLL >R
  LOOP DROP ;
( Compile time, or "early binding", literal definition.
Use as:  #[ bit1 bit2 ... ]#              for autocombining bit names,
    as:  #[ fld1 fld2 ... ]+            for constant structure offset,
 or as:   [ 2 3 + 7 * 5 / ]#            for compile time expression.  )
: #[ 0 [COMPILE] [ ; IMMEDIATE            ( begin compile time literal )
: ]#   [COMPILE] ] [COMPILE] LITERAL ; IMMEDIATE                ( end it )
: ]+   [COMPILE] ]#  COMPILE + ; IMMEDIATE      ( end cmp time offset )
: ]-   [COMPILE] ]#  COMPILE - ; IMMEDIATE      ( end negative offset )
( These words are used to give names to bits.
Usage:
  #bits{ #bit <bit-1> ...  #bit <bit-n> #bits )
: #bits{ 1 ;                 ( begin a series of bit definitions )
: #bit CREATE DUP , 2* DOES> @ OR ; ( n -- 2n   2n ; name bit )
: }#bits DROP ;                ( end a series of bit definitions )
8 CONSTANT BITS/BYTE                  \ number of bits in a byte
: TRANSLATE-TABLE CREATE ( --   TRANSLATE-TABLE name n , ... )
                   DOES> + C@ ;   ( n -- m ; translate a byte )
TRANSLATE-TABLE >MASK  ( bit# -- bit-mask ; return a bit-mask )
  1 C, 2 C, 4 C, 8 C, 16 C, 32 C, 64 C, 128 C,
: BIT[]       ( n -- mask offset ; to index into a bit string )
  BITS/BYTE /MOD SWAP >MASK SWAP ;
: BIT[]@        ( n base -- flag ; fetch truth value of a bit )
  SWAP BIT[] ROT + C@ AND 0<> ;
: BIT[]!        ( flag n base -- ; store truth value to a bit )
  SWAP BIT[] SWAP >R +                \ flag addr <-P R-> mask
  DUP @ R@ NOT AND                    \ turn off addressed bit
  ROT 0<> R> AND OR                  \ OR in flag's truth value
  SWAP ! ;                               \ replace entire byte
: +BIT[] T -ROT BIT[]! ;                 ( n base -- ; set a bit )
: -BIT[] NIL -ROT BIT[]! ;             ( n base -- ; clear a bit )
: ^BIT[]                              ( n base -- ; toggle a bit )
```

```
   2DUP BIT[]@ NOT -ROT BIT[]! ;            \ could be faster...
\ symbols ala Lisp
( Retrieve the unique tag associated with a symbol. If the symbol is not defined,
then create it, otherwise just return its address. )
: $               ( cfa \ $ <token> ; create <token> if needed )
   >IN @ BL WORD FIND          ( save input ptr   find token )
   IF NIP                  ( trash ptr   return cfa if found )
   ELSE DROP >IN ! CREATE                   ( else create it )
   LAST @ NAME> THEN ;                  ( and return its cfa )
: [$]                             (compile time version of $ )
   LAST @ $             ( save so UNSMUDGE won't get confused! )
   [COMPILE] LITERAL LAST ! ; IMMEDIATE    ( fix for UNSMUDGE )
\ Embedded colon defs
( Braces defin "literal words" similar to unnamed LAMBDA expressions in Lisp. The
code>                       : foo bar baz ;
                            : moby ... ['] foo ... ;
may be replaced by ---> : moby ... { bar baz } ... ;
and acheive the same effect without making foo a word too. )
: {            ( --- branch-patch-addr init-state unnamed-pfa )
   STATE @ DUP >R IF       ( begin an unnamed word definition )
     COMPILE branch         ( build skeleton branch around it )
     HERE 0 ,
   ELSE ] THEN
   R> HERE ; IMMEDIATE
: }    ( branch-patch-addr init-state unnamed-pfa -- unnamed-cfa )
   COMPILE EXIT                       ( end definition with EXIT )
   CP @ SWAP PFA, nest JMP, SWAP                \ build code field
   IF SWAP HERE OVER - SEAP !    ( patch offset into branch skel )
     [COMPILE] LITERAL
   ELSE [COMPILE] [
   THEN ; IMMEDIATE                   ( compile cfa as literal 0 )
\ odd exits & tock
( A good old fashioned GOTO is sometimes quite useful. )
: GOTO R> DROP >BODY >S:O >R ;       ( %word - - \ ['] word GOTO )
: GO   ' [COMPILE] LITERAL COMPILE GOTO ; IMMEDIATE      \ GO word
( These words retuirn from the word that called them. )
: 2EXIT   R> R> 2DROP ;                     ( double whammy return )
: ;;   COMPILE R> COMPILE DROP [COMPILE] ; ; IMMEDIATE        \ ditto
: ?EXIT   IF R> DROP THEN ;          ( conditional exit ala muLisp )
( ` , pronounced "tock" does either a : or R: as needed. If tick provides the ad-
dress, tock provides the data. )
: `   >IN @ >R BL WORD R> >IN ! FIND IF R: ELSE : THEN ;
\ Debugging aids
: X.   BL EMIT BASE @ 16 BASE !
   SWAP 4 U.R BL EMIT BASE ! ;                      ( n -- ; hex print)
: .' ' CR DUP  ." cfa " X.                    \ show name & addresses
   DUP >BODY  ." pfa " X.
   BL EMIT >NAME .NAME ;
\ Stolen from C
: |!   OVER @ OR SWAP ! ;            ( addr bits -- ; *addr |= bits )
\ Stolen from from FORTRAN IV
: **   ( k n -- k**n ; raise integer to an integer power )
   1 SWAP 0 DO OVER * LOOP NIP ;
( This word delays execution for the specified number of timer ticks. Since the
need to delay occurs frequently in the source code, and it is isolated here to pro-
vide a single point of change for maintenence reasons.)
VARIABLE #ticks                          \ timer cell
```

```
: ticks-delay   ( n -- )               \ delay n ticks
   #ticks !                             \ initialize ticker
   #ticks TICKER DROP                   \ start ticker
   BEGIN #ticks @ WHILE REPEAT ;        \ wait till expired
: BETWEEN?   ( x i j -- flag )          \ T if i <= x <= j else NIL
   >R OVER <= SWAP R> <= AND 0<> ;
ASCII A CONSTANT 'A'   ASCII Z CONSTANT 'Z'
ASCII a CONSTANT 'a'   ASCII z CONSTANT 'z'
: TO-UPPER ( ^string ^STRING -- ^STRING ) \ convert a string to upper case
   OVER C@ OVER C!                      \ copy length
   OVER C@ 1+ 1 ?DO                     \ copy & convert string
     OVER I + C@                        \ get source char
     DUP 'a' 'z' BETWEEN?               \ is it lower case?
     IF 'a' - 'A' + THEN                \ yes, make it upper
     OVER I + C! LOOP NIP ;             \ put in STRING. &ct.
DECIMAL 13 CONSTANT <CR>               \ carriage return in line delimiter
:.\   <CR> WORD COUNT TYPE CR ; IMMEDIATE \ for messages in INCLUDE files
: D>S   COMPILE DROP ; IMMEDIATE        \ for symetry with D>D
```