ThinLisp Manual

Version 0.5
12 October 1999

The ThinLisp Group:
Jim Allard
Ben Hyde

Table of Contents

Acknowledgements

ThinLisp was begun in 1995 as a late night project born of the frustration of attempting to ship commercial quality software products written in Lisp. Four years later I'm still obsessed by the issues that drove me to it in the first place — practical use of high level languages to implement complex applications while achieving the performance that seems easy when using languages that are semantically "close to the silicon". Hopefully this implementation will lead to some useful insights towards that goal.

Thanks to Ben Hyde for convincing Gensym and myself that this system should be taken off the shelf, polished up, and released as open source software. Mike Colena and Glen Iba wrote significant parts of the implementation. Nick Caruso, Joe Devlin, Rick Harris, and John Hodgkinson and also contributed their time, sage opinions, and code. Thanks to Lowell Hawkinson, Jim Pepe, and Dave Riddell for their support of this work at Gensym Corporation. Thanks to Kim Barrett and David Sotkowitz of IS Robotics, Inc. for contributions of multiple-inheritance code and opinions, and to Rod Brooks and Dave Barrett for their ideas. Finally, thanks to my wife, Gerry Zipser, for putting up with the bleary-eyed aftermath of coding sessions, and for helping to focus efforts during this past summer's release push.

Jim Allard
Newton, Massachusetts
October 6, 1999

Preface

ThinLisp is an open source Lisp to C translator for delivering commercial quality, Lisp-based application. It implements a subset of Common Lisp with extensions. ThinLisp itself is written in Common Lisp, and so must run on top of an underlying Common Lisp implementation such as Allegro, MCL, or CMU Lisp. The C code resulting from a translation can then be independently compiled to produce a small, efficient executable image.

Originally designed for real-time control applications, ThinLisp is not a traditional Lisp environment. It purposefully does not contain a garbage collector. ThinLisp produces compile time warnings for uses of inherently slow Lisp operations, for consing operations, and for code that cannnot be optimized due to a lack of sufficient type declarations. These warnings can be suppressed by improving the code, or through use of lexical declarations acknowledging that the code is only of prototype quality. Code meeting the stringent requirements imposed by ThinLisp cannot by sped up by rewriting it in C.

The ThinLisp home is `http://www.thinlisp.org/` . The newest source distributions of TL and this manual can be found here. Bugs should be reported to `bugs@thinlisp.org`.

1 Rant

1.1 My Excuse for Ranting

I've imagined the reactions people will have when they see that there is yet another Lisp to C translator implementation being released. Mostly I've expected that people would think I'm stark raving mad to engage is a large work, to serve an almost non-existent user base of programmers in a language that is largely discredited while being actively antagonistic towards many of the design principles this language has employed for decades. Since I'm expecting that people will think I'm mad (and half believing it myself), I then feel entitled (even obligated) to engage in a rant to explain the motivations of such a curious endeavor. Please excuse this indulgence and look charitably upon it as a form of self-therapy.

1.2 Lisp Background

More than fifteen years ago I was in a recitation session of an Introduction to Artificial Intelligence course. The lecturer, Patrick Winston, had come in for a session on automated planning. At one point I asked a question about how a certain situation might be handled in the system we were studying, and he answered that it couldn't be handled in that system and that solving that problem was a research topic. The fact that problems as simple (so I thought at the time) as what we were describing were research, and that as a sophomore in college I could grasp them seemed amazing to me then, as it still does now. On the spot I could imagine six different ways to attack that problem, and none of them felt any less plausible than some of the established techniques that we were studying. I was hooked.

At the time, this sort of optimism and arrogance about the immediacy and inevitability of technical success in AI was rampant. The convergence of breakthroughs in techniques, tools, and venture capital money that flowed like wine was intoxicating for everyone.

No one thought this work would be trivial, so the best software development tools available would be required. And of course the best tools from the perspective of the AI community included the language that this community had been developing since the late 1950's, Lisp.

Originally, Lisp was developed as an experiment in providing a direct means of implementing lambda calculus, where operations were first class objects that could be manipulated in the same manner as data. The presumption behind many of the Lisp design choices was that the most important goal was to maximize a programmer's productivity in developing new functionality by enabling the purest form possible of coded solutions, and so to hide as many of the platform specific details as possible. In the ongoing evolution of Lips, considerations of the efficiency of processing speed and memory use came in a distant second to programmer convenience and to the attempt to protect the programmer from bugs caused by mathematically abhorrent behaviors such as integer wraparound.

Perhaps because of this devotion to developer productivity; perhaps because of the synergy between language syntax, parsing, and data; perhaps because of the invention of the Lisp macro facility; or perhaps because it was invented down the hall, Lisp became the

language of choice for research and experimentation in new programming constructs (at least at MIT). Many of the different possible approaches to object oriented programming were first tried out in Lisp. Examples are multiple inheritance, multi-methods, prototype-based object systems, and automatic method combination via whoppers, wrappers, before, and after methods. Other innovations experimented with in Lisp are error handling systems, multi-platform file naming approaches, and guaranteed execution of clean-up code (i.e. unwind-protect).

In 1984 a group formed to attempt to unify as many of the best features of different Lisp dialects as possible into one central standard. From this effort came the excellent books *Common Lisp, The Language* by Guy Steele and it's second edition, which incorporated many of the changes that would eventually be codified into ISO Common Lisp. This is the dialect that dominates Lisp work today, at least in the U.S.

1.3 Lisp in Action

In any case, throughout the 1980's the bulk of the research labs and companies that attempted to commercialize AI technologies used Common Lisp to implement their products. The most broadly deployed AI-based technique was expert systems, also called rule-based systems.[1] Early results were encouraging. In the lab, sometimes on specialized hardware, the software engineers were able to demonstrate the beginnings of the functionality that they were promising.

However, when it came to deploying these systems, things didn't turn out so smoothly. Most software products of that era were coded in languages such as Fortran, Cobol, Pascal, C, and a assembly language. It turned out that Software developers who could breathe fire in Lisp were clumsy, inefficient, or flatly unwilling to port their systems to these traditional languages.

Customers were loathe to buy expensive, special purpose hardware for these applications while cheap, ever more powerful PC computers were rolling out. Stories of this sort are rife throughout the industry, but I'll speak to just the cases I personally was involved in. While the presence of Lisp as the basis of these software systems cannot be fully blamed for their success or failure, in all cases it was a controversial and high-visibility part of the system.

The Mentor system for preventative maintenance was a success story. A group at Honeywell made a forward-chaining, rule-based system that helped novice technicians perform preventative maintenance on centrifigal chillers, essentially large air-conditioning units found in office buildings. Honeywell had a significant business in maintenance services for traditional compressor-based chillers, but it lacked trained personnel to go after this new and lucrative service market. A corporate research lab made a Lisp-based implementation of a run-time environment for Mentor that ran on a ruggedized Grid laptop computers. While the system was useful, it ran sluggishly and the Lisp-based implementation left little room on the laptop for the system to grow and improve. A technology transfer project was begun which resulted in a software engineer within the service division rewriting the application

[1] Of course this is ignoring other, more broadly applicable developments, such as multi-tasking operating systems.

in C, greatly improving its speed and reducing its size, allowing the system to be expanded into a commercially viable tool. As far as I know, this system is still in use.

The Cooker system was a real-time supervisory level monitoring system also built at Homeywell which was intended to be used in process control plants. It was implemented on a Symbolics 3640 which was deployed to a paper mill in Pennsylvania. Though care had been taken to tailor memory use such that a generational garbage collector would efficiently reclaim the vast majority of discarded memory, every 6 weeks or so a hemisphere flip would occur, during which the system could not maintain adequate real-time performance for several hours. After the third time that this happened, the plant crew turned off the system and never restarted it again.

The G2 system is a real-time supervisory level control system built by Gensym Corporation. It's source code was written in Lisp, and for approximately 3 years was released as a world-saved image made from tranditional Lisp environments on UNIX workstation class machines. In 1990, Gensym ported its system to Ibuki Common Lisp, which was a Lisp-to-C translator descended from Kyoto Common Lisp. By translating to C and discarding those parts of Common Lisp that we didn't use, we were able to decrease the size of our shipped product by 25%, a fact that was greatly appreciated by a custombe base tired of constantly having to buy more memory for computers. In 1993 Gensym changed again to a different Lisp to C translator made by Chestnut Software. This produced an additional 40% decrease in size, and a significant improvement in performance. The ThinLisp translator is a philisophical descendent of the Chestnut translator, though ThinLisp was written from scratch. One of the factors cited by the head of sales of G2 was that once we used a Lisp to C translator, he could say that the product was "deployed in C", thereby side-stepping the Lisp issue, which had been a significant downside to our system in the eyes of many potential customers.

The BoreRat is an autonomous robotic device that travels through the piping installed in oil wells. Built by IS Robotics, it is capable of travelling 2 miles straight down into a well, perform maintenance and data logging tasks, and then return to the surface, all under its own power and control. The software for this system was written in L, a Lisp subset made for embedded processors, in this case a Motorola 68376. The first prototypes of the BoreRat became operational in 1999 and are currently being marketed to the oil field services industry by IIC.

1.4 Mathematician's Languages vs. Engineer's Languages

From the beginning Lisp was a mathematician's language. To understand what I mean by Lisp this, I must explain my view of the differences between the goals of scientists, mathematicians, and engineers.[2]

The goal of scientists is to learn truths about the world around them, using observation, analysis, hypothesis, and experimentation to advance and buttress their claims of discovery. Pure science aspires to being unconcerned with the utility or applicability of the information they seek. The ultimate goal is to discover what *is*.

[2] All apologies in advance to my father-in-law (a scientist) and my brother-in-law (a mathematician).

Mathematicians aspire to an even higher goal — understanding what truths there are or that there might be, regardless of or even entirely divorced from the limits of the physical world. The studies of mathematicians are the stuff of pure thought and imagination, yet bound by the strictest of rules of proof. To build robust castles in the air based on proofs combining into a single monolithic whole requires that all pieces must act perfectly in concert. Therefore the mathematician must constantly be vigilant against contradictions between the different portions of his whole proof, where one technique carries with it constraints which must be respected within all other parts of the proof. With this rigorous technique in hand, the ultimate goal is to discover what is *true*.

On the other hand the goal of the engineer, in the abstract, is to bend the behavior of an uncooperative world to his will. Whether it is the stereotypical train engineer, civil engineer, acoustic, electrical, mechanical, or software engineer, their work is to use the fruits of science and mathematics, exploiting practicality and expediency, to operate or design a physical entity so that it performs a specific task, usually with the motivation of the almighty dollar. Within the constraints of the physical world, the ultimate goal is to accomplish what is *practical*.

Within the computer science community, examples of personalities suited to each of these different disciplines abound. Within the the small community of computer language designers, the mathematician and engineer distinguish themselves through two very different approaches to language design.

The mathematician espouses denotational semantics, where the behavior of an operation is determined by what the operation *means*. In this style, the goal is to make a complete, and unified system where each element of the language must be intricately coordinated with each other part, so that each element can retain its meaning when used in conjunction with any other part. The semantics of each operation are primary, and the consequences of the definition of one operation can strongly affect the implementation of many other operations. While the behaviors required of any individual operation may become complicated, the programmer using such a language can work confident that his code may be intermingled in complex ways with other's code without fear of each violating the other's assumptions about eventual program behavior. When finished, systems defined along these lines make powerful and elegant environments.

The engineer espouses operational semantics, where what each operation means is determined by what it *does*. In this style, the goal is to determine the elements of a toolbox and to specify the behavior of each, largely independently of any other operation in the language. The definitions of operations stand alone and are designed specifically to minimize the interaction between operations. The semantics of each operation are thin, but this allows the programmer to work confident of the performance and size of the program being generated. Here one is seldom surprised by the behavior of a line of code, and moderately experienced programmers can accurately predict the behavior down to enumerating the machine instructions that are likely to be executed.

Most computer languages is use today are exemplars of the engineering style, defined with operational semantics in mind. Building primarily from the operations that could be implemented on the computing devices that could be made at the time, and perhaps secondarily on the needs of a user base at hand, languages were designed to allow people to command computers to do things. Typically the operations in the language were semanticly

thin (or shallow, if you prefer), to the extent that a programmer of medium experience could almost predict the number of machine instructions required to carry out any given command in the language. Fortran, Pascal, Basic, and C are all examples of languages in this category, with C (and it's derivative, C++) now being the default choice for people building big systems.

Lisp and other Languages (such as Smalltalk, Eiffel, and recently Java) were developed along along a different tack. The operations in these languages were defined with semantics that are deep (or fat if you prefer). Here the focus was not on what an operation *did* but on what it *meant*. Of course there had to be implementations of these operations, but the actual machine instructions carried out were allowed (even expected) to vary widely depending on the manner and situation in which it was used.

An example of how this difference in design perspective plays out can be found in the behavior of the `goto` statement in C and the `go` special operator in Common Lisp. Consider the following two code fragments.

In C:

```
int find_big_int(int *set, int start, int end) {
  Iterate through the set {
    if test
      goto bad;
      ...
  }
 bad:
  ...
}
```

In Lisp:

```
(defun find-big-int (set start end)
  (tagbody
    (Iterate through the set
      (if test
        (go bad))
      ...)
    bad
    ...))
```

In the C code above, regardless of the surrounding code and environments, a user can predict the machine's behavior at the `goto` command and what the likely cost is. In particular, a conditional branch instruction will be generated based on the result of the test expression, and it will immediately begin executing the code following the label `bad`.

In the Lisp code, though the situation appears similar, the resulting behavior may be very different. If the code that implements the "Iterate through the set" element above is a simple `do`, then the behavior will be very much like what happens in C. However, if the iteration is performed using a mapping function, and the `(go bad)` form is contained within an embedded `lambda` form defining a different function, then the situation is very different. Because goto tags have lexical scope, even across function boundaries, requires that the `go` be able to exit the function it is in and re-enter the outer function at the appropriate

point. In this case, entering the `tagbody` statement would require establishing a restart point similar to a `setjmp`, and the `go` would become something similar to a `longjmp`.

This is the kind of performance "gotcha" that frustrates programmers who look at languages from an operational semantics point of view. Attempting to allow the benefits of deep semantics while preventing performance surprises is the underlying goal for ThinLisp.

1.5 ThinLisp's Compromise

One of the goals of ThinLisp is to implement the deep semantics of Lisp, while offering compile time warnings whenever these semantics force behavior that deviates from expectations consistent with thin semantics. If the programmer really intends to use the full, deep semantics, then a lexically apparent declaration can be used to suppress the warnings and accept the more expensive behavior required by the deep semantics.

There are three main areas where ThinLisp warnings are used to help prevent performance gotchas.

- consing,
- run-time dispatched operations, and
- non-local exits.

2 Original Design Notes

The following were the original design notes about what should be built into GL, which became ThinLisp. -jallard 10/12/99

2.1 Def-System

In order to get the flexibility that we want with this development environment, a def-system implementation will be built-in to GL. This is necessary to get the desired layering of libraries and to enable identification of call-trees, module interdependencies, and dead-code.

The base system will be GL. It will contain only the smallest set of primitives possible, though this is still a substantial set. Support for all defining forms and special forms must be built here. There will several systems that implement portions of Common Lisp that we think may not be necessary for all products we intend to build in Lisp. These should include many of the utilities that we currently have in G2 to replace consing versions of the same from Common Lisp. For example, the gensym-pathnames and gensym- streams code should be made into a separate file-io system that may or may not be included into products (G2 and TW need it, GSI does not).

Within the Def-system for a group of files there should be a flag indicating whether or not this set of files is to be distributed as a library or an executable. If it is a library, the def-system should include (or there should be a separate form for declaring) those functions that are called by users of the library, and the def-system should also include (or there should be a separate form for declaring) those call-back functions provided by users of the library that the internals of the library will call (this is the GSI call out case). Foreign functions should be handled as they are now using the def- gensym-c-function form.

2.2 Overview of Translations

There are four different modes in which GL (Gensym Lisp, oops, er, Language) source code will need to be processed, two Lisp compilation modes and two translation modes. The first is development compilation in an underlying Lisp environment. The second is a macro definition compilation to define macros in preparation for a translation. These are otherwise unremarkable uses of the underlying Lisp implementations compile-file and load functions, distinguished only by what we do with the :development and :distribution compiler switches. The third is a translation pre-process step in which the names and data types of variables and functions are determined. The fourth and final is a translation mode in which C files and H files are generated.

The translator from GL to C will be written to make as few dependencies as possible on the implementation it is layered on top of. (As this is being written Lucid has gone bankrupt and been acquired by Harlequin and Apple is dumping off MCL to a third party to support.) In order to keep things portable we will implement our own macro-expander to avoid the problem of differences in macro-expansion environment representations. Those Lisp functions whose argument lists are more complex than positional and optional arguments will be implemented as macros. These macros will expand into functions with fixed

and optional arguments. We will write all of our own macro implementations of all non-function Lisp primitives so that we can control the base set of functions and special forms that these macro-expansions will bottom out within in a portable way. All of these base functions that the translator natively supports will be implemented in the Lisp development environment so that the development environment execution will be as similar as possible to the final C execution, but those facilities that are required to come directly from the underlying Lisp during development (e.g. special forms) will be had through conditional macro-expansion that uses the underlying Lisp. For example, a call to gl:list with 5 arguments will always macro-expand into calls to the gli::list-5 function. However, calls to gl:catch will expand to gli::catch while translating and will expand into lisp:catch during development. The special-form gl:the will expand such that it performs type checking at development run-time and type declared local variables will at least have their initial values type-checked. All reasonable effort will be made to avoid redundant run-time checks in development.

A Lisp function fragment illustrating the outermost loop of a translation is as follows. The actual implementation will need further complications, such as recompile arguments.

```
(defun translate-system (system)
  (loop for used-system in (system-used-systems system) do
    (load-system used-system :binaries t :gl-preprocess-info t
                 :gl-trans-info t))
  (loop for file in (system-files system) do
    (when (binary-out-of-date-p file)
      (compile-file file))
    (unless (binary-loaded-p file)
      (load-binary file))
    (when (gl-preprocessed-date-out-of-date-p file)
      (gl-preprocess file))
    (unless (gl-preprocessed-data-loaded-p file)
      (load-gl-preprocessed-data file)))
  (loop for file in (system-files system) do
    (when (or (gl-trans-data-out-of-date-p file)
              (gl-c-file-out-of-date-p file)
              (gl-h-file-out-of-date-p file)
              (progn (load-gl-trans-data file)
                     (gl-trans-data-indicates-retranslate-p file)))
      (gl-translate-file file)))
  (gl-dump-makefile-information system))
```

2.3 Compilation Passes

The development compilation pass can be detected by the :development switch in *features*. The macro compilation pass has no significant *features* included. The pre-process translation pass and the translation pass can be detected with the :translator and :no- macro switches in *features*. Note that the pre-process pass may be interleaved with the macro pass (i.e. we may macro compile then pre-process each Lisp file in turn), but these features can still be reliably used to detect the difference. Also note that we will likely want to perform development translations which will produce images with full runtime type checking,

which could be detected by having both :translator and :development available at the same time. This may be difficult, since the :development switch has often been confounded with dependencies on Lisp development environment (i.e. :lucid and formerly :lispm) switches. An alternative but inferior approach would be to use a different switch to control translator type-safety checks.

2.4 Pre-Process Compilation Pass

The pre-process pass is used to gather information about the following: function name, argument types, and return types needed to translate calls to this function; and variable data types needed to compile fetch and set references. This information must be gathered before translation begins on any files, since Lisp generally allows forward references to as-yet undefined functions and variables. This pass over each file must occur after that file has been compiled (if necessary) and loaded during the macro compilation pass. (Consider collecting this information during the macro-expansion of gl:defun, gl:defvar, etc. macros.) During this pass all top-level forms in a file are read and macro expansion proceeds on each form only far enough so that top level defining forms can be examined and their types determined. Normally this would only be far enough to example declare statements on functions and global declaim statements. In practice, we may also attempt to perform limited macro-expansion on the bodies of functions that do not contain argument and return value type specifications in order to determine stricter typing information, but I'm expecting that this may be too time-consuming to do without some limitations, for example stopping after 100 calls to gl-macroexpand-1.

The gathered information will be stored in properties of symbols and will also be written out into a .glp file (GL pre-process, a Lisp syntax file suitable for processing with load, written into the macro directory next to the binary) for each .lisp file pre- processed. This enables us to reload the .glp file during subsequent pre-processing in which the source Lisp file has not changed.

2.5 Translation Compilation Pass

The translate pass is finally used to read source Lisp files and translate their contents into C and H sources files. Besides the translated functions and variables, each C file will also contain an init function, which will allocate code constants needed by the translated functions and perform the operations called for by the top-level forms of the translated file. Some constants will be globally shared, such as compiled-function objects and symbols. At the end of each translation, a .glt file (GL translation data) will also be written out containing information about which globally optimized constants are used, which of those are initialized within a translated file, and which functions and variables are referenced with what type signatures. This is needed for later incremental translations. During these later translations it may be necessary to retranslate due to changes in prior initialization of globally shared constants, such as when a symbol is no longer initialized in a prior file, and so must be initialized in this file. Similarly, when translating calls to functions, information about the type signature of the called function is used to translate the call site. If the type signature of a function changes, that will force a retranslate of files containing calls.

Lastly, we have always been concerned about changes to Lisp macros being propagated to all callers. If during translations we keep track of which macros are called and keep a checksum of the compilation of its macro-function, we may be able to compare the checksums of later translations and automatically determine when retranslates must be performed.

2.6 Packages

Packages should be designed as follows. The AB package should now use the GL package to get all of its underlying Lisp implementation symbols. The GL package will not use any other package and it will export all GL symbols that we use. The translator functions will be available at translation time through the GLT package, and the translator itself will be written within the GLI (GL Internals) package. During development, some GL symbols will have been imported from the Lisp package, but at runtime there will be no Lisp package, only GL.

During runtime, the AB and GL packages will be available, but the GLT and GLI packages will be absent. Note that it is currently my intent to have fully reclaimable package and symbol structures so that new packages using the AB package can be created at runtime and symbols removed from packages using the unintern function will in fact be reclaimed as they are uninterned.

There is support in the symbol data structures for importing, but that will not be used at this time.

2.7 Stacks

A global Obj_stack and Obj_stack_top will be maintained for holding shadowed values of global variables and setjmp environments for catch and unwind_protect scopes. The stack needs to be structured so that it can be searched from the top down towards the bottom finding catch tags for throws, unwind_protect scopes to execute, values to reinstate back into global variables, and multiple_value sets to reinstate back into the multiple_values_buffer. Each function that pushes values onto the stack must reset the Obj_stack_top back to it's original value before exiting. Catch and unwind_protect locations must cache the Obj_stack_top value available on entry to their scope and be prepared to restore it. When throws are performed, stack unwinding will determine appropriate catch locations for throws, restore global variable values whose binding scopes are exited, and execute unwind-protect cleanup forms.

2.8 Global Variable Binding

Bindings of global variables will be handled as follows. The current value of the global will be stored in a local lexical variable of the binding function and also will be pushed onto the Obj_stack. Next a pointer to the value cell of the bound variable is pushed onto the stack, and finally the keyword :global-binding is pushed. The global variable is then set to its new bound value. If a normal lexical exit is made from the scope of the binding, then the global variable's old value is restored from the location in the local lexical variable of the binding function and the three values are popped off the stack by decrementing Obj_stack_top by

3 words. If a non-local exit is made from this scope, then unwind_obj_stack will recognize the keyword :global-binding and restore the value into the symbol-value cell pointed to on the stack.

2.9 Unwind-Protect Scopes

On entry to an unwind protect scope, a call to setjmp is made with a jmp_buf environment in order to save this location within the C calling stack. This jmp_buf (which is a pointer, see "C, A Reference Manual, 3rd Edition" by Harbison and Steele, p. 352) is pushed onto the Obj_stack, then the keyword :unwind-protect. Any unwind scopes will then perform a longjmp to this jmp_buf as the stack is unwound. After performing the cleanup forms, unwinding should then continue in this case by performing another longjmp to the next unwind location on the stack, or to the target catch tag if no further cleanups are needed. On a normal exit from this scope, Obj_stack may merely be decremented, abandoning this cleanup on the stack. Note that all local lexical variables in scope around the unwind-protect must be declared volatile in the translation.

2.10 Catch and Throw

On entry to a catch form, a setjmp is performed with a jmp_buff environment. This jmp_buf, the catch value, and the keyword :catch are then pushed onto the stack. If a throw is performed, then a longjmp will be executed with the appropriate values held in the Multiple_values_buffer. The receiver of values from the catch form then may or may not unpack values from that buffer. Note that all local lexical variables in scope on entry to the catch form should be declared volatile in the translation.

2.11 Multiple Values

Within locations where the source and receiver of values are mutually visible, then all single valued expressions will be handled as C expressions, and all multiple valued expressions will be handled with a series of setqs into gensymed variables. However, translations of locations where the source of values is a function of unknown type signature or where multiple values are returned to an unknown receiving location, then a general protocol of multiple values will be used. In these cases, multiple values will be handled with a global Multiple_values_buffer and Multiple_values_count. The first Lisp value will always be returned as a normal C expression or function value, with any secondary values being placed into the Multiple-values-buffer. If no values are returned, then the values count will be set to 0 and NIL is returned, which is the default value when receiving more values than are supplied. Sites receiving only the first value may use the value of the C expression. Sites receiving more one value must check the C values count, retrieving values from the Multiple_values_buffer, or NIL if more values are desired than are supplied. In multiple-value-prog1, catch, or unwind-protect cases, then a protocol will be supplied which enables all values being returned to be cached onto the global Obj stack. After all cleanup forms in the multiple-value-prog1, catch, or unwind- protect have been executed, then the cached values should be popped off the stack and back into the multiple-values-buffer.

2.12 Constants

Constant values of type char, sint32, fixnum, and double will be translated into inline C constants. The sint32 and fixnum constants should be able to be emitted without a trailing L (as opposed to current practice) since we will not be using the AlphaOSF long type, and so we expect to not be subject to the preprocessor type conversion bug that led us to adopt the L integer suffix rule.

Constants of type string, symbol, and compiled-function will be implemented using initialized C struct variables, and these will be constant folded across all translated C files. Translated Lisp references to these constants will actually be references to the address of these C variables. For string constants, typedefs will be emitted into the C files for string structures containing differing lengths of string. Where the initial value of a string constant is longer than 80 characters, the initial characters of the string will be initialized using the array variable initialization format instead of the normal string constant approach. This eliminates the long lines that break some compilers (e.g. VAX C). Variables for strings should be declared as "const". Symbol and compiled- function variables cannot be declared const, since they must be modified during initialization to intern symbols, install compiled-functions in symbols, and install pointers to C functions in compiled-functions.

The number of external symbols needed within individual C files is a concern, since we have seen this be a link time limitation on some systems. If this is the case, then we can still achieve global constant folding within a translated system by using an approach where the pointers to constants are installed into a single large constant vector, with indices into this vector used to fetch the addresses of initialized constants. This approach requires a more complicated scheme for use of constant vector indices that I would like, but it can be handled without causing frequent recompiles of users of constants by reserving indices and only renumbering constant indices on declared retranslates of a system.

2.13 Streams

I'm hoping that most support for Common Lisp streams can be avoided in the first implementation of the translator. We already have an implementation for complex formatting, and so I expect that the only need for streams will be to print to standard output.

2.14 Variables

Global Lisp variables and parameters will generally be represented as C variables of types Obj, sint32, and double. These C variables will represent the symbol-value cell of the actual Lisp symbol. In cases where the Lisp symbol for a global variable also exists, the symbol-value cell of that symbol will be a pointer to the C variable's location, and the local_value bit of the symbol will contain 0. In cases where there is a Lisp symbol for a C variable, the C variable will always be of type Obj.

2.15 Functions

Each Lisp function will be implemented as a C function. The C function will accept positional arguments for all required and optional arguments of the Lisp function. Keyword

and rest arguments are not supported at runtime, though we may provide support for keywords on functions as long as no attempt is made to generate a compiled-function object for that function (a prerequisite to funcalling).

2.16 Translator Data Structures

The following sections present designs for the data structures and functions required in Lisp for the translator. They are presented in no particular order, other than being fairly bottom up. The translator will make use of defstruct for all structures.

2.17 C Name Spaces

A part of translating is determining the C identifiers to be used to represent Lisp symbols within the translated files. A C namespace structure will be used to represent each of the layers of C namespaces; global scope, file scope, and function scope. C namespaces keep hashtables relating C identifiers delcared within a scope to short descriptions. The descriptions will be lists whose first value is one of the symbols function, variable, macro, or reserved. For variables and functions, there is a second element in the list which is a string containing the appropriate extern declaration to declare the type of that identifier.

```
(defstruct (c-namespace
             (:constructor make-c-namespace (surrounding-c-namespace?)))
  (local-identifiers (make-hash-table :test #'equal :size 1000))
  (surrounding-c-namespace? nil))
```

There will be a global C namespace that contains all built in identifiers and reserved words of C. This set will be determined by numbly entering all identifiers listed in the index on pages 385 through 392 of Harbison and Steele, all identifers built-in to GL in the hand-written C code, and any POSIX identifiers we use. Omissions in this list will show up as multiple definitions or improper links only if Lisp symbols are made that happen to collide with the C versions. In all cases, errors of this form should be able to be fixed up by adding identifiers to the global namespace.

The function 'c-namespace-get-identifier' takes a string and returns the data associated with that identifier. It may be used with setf. Note that getting an identifier will search through surrounding namespaces, but setting will directly set the given namespace.

```
(defun c-namespace-get-identifier (c-namespace identifier-string)
  (loop for namespace = c-namespace then next-namespace?
        for next-namespace? = (surrounding-c-namespace? namespace)
        while next-namespace?
        finally (gethash (local-identifiers namespace)
                         identifier-string)))

(defsetf c-namespace-get-identifier set-c-namespace-get-identifier)

(defmacro set-c-namespace-get-identifier
  (c-namespace identifier-string declaration-info)
  `(setf (gethash (local-identifiers ,c-namespace) ,identifier-string)
 ,declaration-info))
```

As translation proceeds, all external symbols of each translated file should be added to the global namespace. When a namespace is made for a file, it should not inherit symbols from the global namespace, except as they are imported through extern statements. When a namespace is made for a function, it should inherit from the file namespace.

Note that no identifier within any of these namespaces should exceed 32 characters in length, since that is the limit for the VAX linker.

2.18 C File Structures

During translation of a Lisp file to a C file, a C file structure will be held in a global variable. The C file structure will contain an output stream for the C file, an output stream for the H file, a namespace, a data structure for referenced constants, and an initialization C function. It will also have a pointer to the Lisp module symbol name, which will have properties holding the information from the .glp and .glt files.

The following functions are available for C file strucutres; make-c-file-structure, emit-type-declaration-to-c-file, emit-variable-declaration-to-c-file, emit-function-to-c-file, emit-constant-to-c-file, emit-string-to-c-file, emit-character-to-c-file, emit-line- comment-to-c-file, emit-indentation-to-file, emit-freshline-to-file, and emit-newline-to- file. The function reclaim-c-file-structure will also be written, but it may only destruct the file structure enough for the garbage collector to take care of it.

The emitters for the differing structure types will be described below. The emitters for indentation, strings, characters, newlines, and comments are implemented on the file structure so that we can take care of inserting line breaks into long expressions and to enable comments to be inserted at the ends of lines. This is to be used to comment frame and structure slot accessor expansions, for example. When an expression or character is to be emitted into a file, a check is made to determine how far into the line we already are. If the number of characters emitted into the line so far plus the accumulated comments are longer than some limit (probably 80 characters), all pending end of line comments will be written to the output stream, a newline will be sent, the current indentation level plus two will be emitted into the new line, and then the new string or character will be sent. We can imagine fancier pretty printers for C expressions that would get argument indenting correct, but not today. When a comment is emitted to a C file, it will be kept in a list and will be flushed out to the line before any newlines are emitted to the file.

2.19 C Type Declaration Structures

In some cases, new C typedef statements may need to be emitted within a C file. In particular, I'm thinking about the case of C constants that contains arrays of varying sizes such as strings, simple-vectors, byte-vectors, and double-float-vectors. If such things are needed then this structure will be used to represent them.

2.20 C Variable Declaration Structures

Variable declaration structures will be used to hold information about all C global variables. They will also be used to hold information about Lisp constants that are emitted

as "const" C variables. Lists of these will be included in .glt files. For global variables, a property will be held on the Lisp symbol pointing to the C variable declaration structure. That strucuture will contain the C identifier, C type, and C initial value. Note that some global variables will also have initial values installed within C file initialization functions. Perhaps there will be a need for a flag indicating this in the C variable declaration structure, but I'm not certain of that.

2.21 C Function Structures

C functions will be represented during translations as structures that contain a return type spec, a list of argument type specs, a C name string, a Lisp name symbol,a list of C local variable names and types, a list of available C local variables and types, and a compound statement body. During the translation of a function, a global variable will point to the function. Functions will exist to allocate and reclaim C variables for the function. When a Lisp variable goes out of scope, the C variable for it should be reclaimed for possible reuse later in the C function. Reuse should depend on the type of the variable and on the desired C name. In an earlier version of Chestnut they only depended on type for reuse, and many people complained that the resulting code was unreadable.

2.22 C Statement Structures

For each kind of C statement, there will be a different C statement structure type. These structures will all include the type c-statement, which contains one a slot, an emitter compiled function.

The function emit-statement-to-file is used to write all forms of C statements to the output streams of file structures. This function takes a statement, a Lisp stream, and an indentation level. It dispatches to a different emitter for each kind of C statement, using the documentation in "C, A Reference Manual, 3rd Edition" by Harbison & Steele, pages 213-235 as a formatting guide.

Compound statements are central to translating, so they have some extra operations for receiving statements, declarations, and labels into them, but otherwise they are unremarkable sibling types to all of the other C statement types. As a matter of convention, each C statement emitter will be responsible for statement delimiters and the newline after itself. In general, statement emitters may not assume that a newline has preceeded them, since conditional and iterative statements emit their held statements without first emitting a newline, so that the open brace is on the same line as the condition, per C coding conventions. However, the function to emit indentation will always call emit-freshline-to-file before emitting indentation, so in practice emitters should not need to send newlines before their indentation.

Each statement emitter is responsible for its own indentation. Compound statements should emit their held statements at the indentation level given, and their closing curly brace at the given level minus one. All other statements also emit themselves at their given indentations. Conditional and iteration statements are responsible for incrementing the indentation level passed to their contained statements.

The following kinds of C statement structures will be implemented: c-compound-statement, c-expression-statement, c-label-statement (yes, even though it's not really a statement), if-statement, while-statement, do-statement, for-statement, switch-statement, break- statement, continue-statement, return-statement, and goto-statement. Note that there is no null-statement structure, and that this is the only C statement without a corresponding structure. Null statements may be emitted as parts of compound-statements, but otherwise are never specifically called for.

2.23 C Compound Statement Structures

C compound statements will be used to hold all emitted C statements, statement labels, and lexically scoped variable declarations. When a translation is occurring on a Lisp function, the function that performs a translation of a Lisp expression will be given a Lisp s-expression, a required type spec for the result of the translation, a prolog C compound statement, and an epilog C compound statement. The translation function will return a C expression structure that returns the values of the translated Lisp expression, but it will also emit statements into the prolog C compound statement which must be run before the C expression is evaluated, and it will emit cleanup statements into the epilog C compound statement which must be run after the C expression is evaluated.

The following functions will be available for C-compound-statements:

- make-c-compound-statement,
- emit-statement-to-compound-statement, and
- emit-label-to-compound-statement.

Make-c-compound-statment is an argumentless constructor. The most commonly used function will be emit-statement-to-compound-statement, which is used to add a new statement to the end of the sequence of statements currently held in the compound-statement. Note that if the statement being added is itself a compound statement, then the effect is to concatenate its sequence of statements to the end of the containing compound statement. This eliminates unneccessary nested curly braces, and is a needed memory optimization given how heavily I intend to use compound statments when translating.

If the compound statement contains no variable declaractions or statements, then it will be emitted as a single semicolon and a newline. If the compound statement contains no variable declarations and a single contained statement that is neither a c-conditional-statement nor a label-statement, then it will emit a newline and its one statementat the given indentation. Otherwise, it will emit a curly brace, a newline, emit all variable declarations at the given indentation level, emit all its held statements and labels at the given indentation level, emit a null statement if the last thing emitted was a label, then emit its final curly brace at the given indentation level minus one.

Note that statement labels will be emitted into compound statments just as C statements are, except that they will be preceeded by one fewer level of indentation, and if the last thing within a compound statement is a label, a null statement (i.e. a line containing nothing but a semi-colon) is emitted after the label, but before the final curly brace of the compound statement.

Also note that conditional statements held within compound statements will always be enclosed within curly braces to protect them from inappropriately acquiring dangling else clauses from potentially surrounding conditional expressions.

2.24 C Expression Statement Structures

Expression statements will hold a c-expression structure. Expression statements will be emitted by emitting the appropriate amount of indentation, then emitting the expression, a semicolon, and a newline. Note that if the expression is side-effect free, then this emitter may decline to emit anything. C compilers are allowed to make this optimization, so this translator is allowed as well (H&S 3rd, p. 215). Also note that if the expression is a c-conditional-expression, then this expression should be transformed into a c-if- statement, with the then and else clauses becoming new c-expression-statements. Typically these C expression statements will be assignment, increment, decrement, or function call expressions.

2.25 C Label Statement Structures

C labels for goto tags, switch case values, and switch default branch points will be represented via a c-label-statement. The label slot of this structure will contain a string, which be either a goto tag string, the string "case <x>" where x is a switch dispatch value, or the string "default". The emitter for a label will emit the given indentation level minus one, emit the held string, a colon, and a newline. This indentation level sets labels even with the opening curly braces of their containing compound statements, and sets switch case values and default tags at a level even with the opening switch statement.

2.26 C Conditional Statement Structures

This statement contains a list of alternating c-expressions and c-statements. Each expression is a condition for the following statement in a multiway conditional statement (H&S 3rd, p. 218). If the final statement for the multiway conditional is a simple else (i.e. always perform the last statement if no preceeding clause was selected) then a NIL should be found in place of the c-expression in the last two elements of the list. A c- conditional-statement emits the given indentation, emits the string "if (", emits the first c-expression, emits the string ") ", then emits the corresponding c-statement at the given indentation level plus one. For all subsequent pairs, it should first emit the given indentation level and the string "else ". If the c-expression is non-null then emit "if (", emit the c-expression, and emit ") ". Then emit the c-statement at the indentation level plus one.

Note that if the C statements within the conditional are compound statements, then their opening curly braces will be emitted on the same line as the expression. If it is any other kind of statement or is a compound statement holding a single statement, the act of emitting its indentation will force a fresh line, placing the statement alone on the next line, one indentation level further in from the if statement.

The proof that we will not experience the dangling else problem is only valid if all statements held within conditional statements are emitted as compound statements. This emitter should verify that by checking for held conditional statements and wrapping them in a compound statement if any are found.

2.27 C While Statement Structures

The c-while-statement will contain a c-expression and a c-statement. It will emit its given level of indentation, the string "while (", the c-expression, the string ") ", and then emit the c-statement at the given indentation level plus one.

2.28 Emitting Indentation

The function emit-indentation-to-file takes a number of indentation levels to emit and a c file to emit into. This function will emit a freshline to the file then emit two spaces of indentation per level, except if the number of levels exceeds 8 (both of these values will be parameters in the implementation). In order to prevent excessively long lines, indentation levels over 8 will have a comment at the beginning of the line saying how many levels of indentation, and then further spaces will be emitted to give an effective indentation of 8 levels. The indentation level will also be stored into the c-file structure, so that line continuations of expressions can be indented to this level plus one or two more extra levels.

2.29 Post-Macro Translation

After macro-expansion is complete, the s-expressions of a function will consist entirely of GLT package special forms and translatable functions. The process of translating is split into several steps, each being kept small enough to be simple, but together providing all the complexity we need to get the quality of translation desired. The following steps are carried out in sequence: structurizing s-expressions and variable references into structures we can use to store information about the s-expressions and variables they came from; flow path identification through block and unwind-protect forms; Lisp type propagation into the structurized forms including expression and continuation analysis, C type choosing for variables and structurized forms.

2.30 Structurizing

Once Lisp expressions have been fully macro-expanded into special forms and function calls, passes are made over the s-expressions replacing s-expressions with an operation structures and then filling out the values within them. The operation structures contain the original operator symbol, an argument list, a continuation pointer, a C return type for the form, a C operator (if the Lisp operator is not a special form), and a flag indicating if this operation and its arguments are all side-effect free. All references to variable values will be replaced by references to the lexical-value-operation and global-value-operation structures. Special forms will have their own subtypes of the operation structure, containing extra slots for that form. For example, the let-operation structure will hold C variable structures for bindings in a LET.

Structurizing is carred out by the function structurize-s-expression. It operates as follows. If the expression argument is constant-p, it is evaled and returned within a quote-operator structure. If the argument is a symbol, then a matching lexical binding is searched for within current-lexical-scope or a global variable binding is found, it is saved in a symbol-value-operator structure. If no lexical or global binding can be found, issue an error warning

and stub in a global variable structure of type T. If the expression argument was neither of the above cases, and is not a cons with a special form or function naming symbol in the car, issue an error warning and return a stub global variable symbol-value-operator to a variable of type T.

If the symbol in the car of the s-expression list is a special form, funcall the structurizer property of the symbol, passing the whole s-expression and storing the result. If the car is a symbol naming a function, structurize each argument, collecting them into a list. Next make an operator structure containing the argument list, the original operator symbol and the Lisp definition structure for that symbol. Next, if the operation is functional (i.e. side-effect free and bases its results solely on arguments, e.g. +, -, car, and svref, but not get, which uses mutable substructure of the symbol) and if all arguments are supplied by quote-operators, then the quoted-values of all the quote- operators should be gathered in a list, and the operator symbol applied to the list. The result should be placed into a quote-operator to replace the previously structurized form. Next, if the Lisp definition of the operation is side-effect free or functional, and if all arguments are also marked as side-effect free, mark this structure as side-effect free.

During the structurizer functions for special forms that produce a new lexical variable binding, variable-binding structures are created and pushed onto the list of bindings for that symbol in its :lexical-bindings property. References to these variables, from either the set-operator or the symbol-value-operator, should examine the symbol's :lexical- bindings property for the first binding structure, store a pointer to it, and place themselves onto a list of references to the variable within the variable-binding structure (these lists are used for variable elimination optimizations and type proofs). After structurizing the subforms, the structurizer that bound the lexical variable should pop it's variable-binding structures off the symbols' properties.

Whenever a structurizer establishes a new global variable binding, a global-variable-binding structure is created and pushed onto a new binding of current-lexical scope. Return statements that exit the scope must unbind the global variable. Whenever a block special form is structurized, a block structure is pushed onto a new binding of current- lexical-scope. These structures are used to determine how far the lexical scope must be unwound when returning from a block, and they are used to determine which block is to receive a return-from. Lastly, whenever an unwind-protect special form is entered, a protect structure is pushed onto a new binding of current-lexical-scope. This is used to determine which protect cleanup forms might be "in the way" of a return-from and require processing before the return can complete.

All of the global-variable-binding, block, and protect structures in current-lexical-scope are needed to handle return-from forms. Structurizing of return-from forms will be handled as follows: A linear search is made for the first block matching the target of the return-from. Each global variable binding encountered during the search will be kept in a list on the return-from structure, so they can be unbound before the return-from exits. If a protect is found before the matching block, then control is going to be passed to the protect instead of the final target block. This is accomplished by creating a return-path structure in the unwind-protect-operation and returning to it. After the protect forms have been completed, the unwind-protect will continue to return towards the target block, passing through the values given to the return-path. Even though control from this return will pass

to the protect, the search continues for the matching block. Once found, the return-from-operation structure is pushed onto the return list of the block. This is needed for type progation in later passes. If a matching block is found in the current-lexical-scope before any protects, then in the return-from -operations structure a pointer is kept to it and the return-from-structure is pushed onto the list of returns to the block.

The structurizers for flet, lambda and labels should gather pre-pass data on the lexically held function, and push the missed lambda forms onto a list of sub-functions. After structurizing is finished on the current function, all subfunctions will be structurized, translated, and emitted into the C file before the enclosing function is translated and emitted. After the defining form is pushed onto the list of subfunctions, the lisp- definition structure for the new function will be pushed onto the :lexical-functions property of the naming symbol. After the body of the flet or labels has been structurized, the definition for the subfunction should be popped off the :lexical- functions property list for the defining symbol.

2.31 Type Propagation

After structurizing has been completed on the body of the function, then the structurized form is passed to the function propagate-types. Propagate-types take a structurized expression and a type specification of the required return type as determined through type declarations or through type proofs made during the pre-pass phase. This function propagates Lisp types and optimized C types. The C type optimizations are fixnum to Sint32, double-float to double, character to unsigned-char, simple-vector to Obj*, string to char* (note there is no difference for fill-pointered strings), (simple-array double- float) to double*, (simple-array fixnum) to Sin32*, (simple-array (unsigned-byte 16)) to unsigned-short*, and (simple-array (unsigned-byte 8) to unsigned-char*. The optimized types will be represented during translation by the symbols just given, but of course will be translated to C syntax when translated. The given optimizations will be performed on all variable-bindings (including global variables) that are declared or proven to contain the named Lisp types. The forms that define translatable operations can be declared to receive the optimized types or standard Lisp types.

Type propagation is iterative at the top level. Upon first receiving being called, type propagation is performed as is. They further type proofs are carried out to see if more tight typing can be had for local variables. If so, then type propagation is performed again, checking again if better types can now be determined for local variables. This process repeats until no further variable type optimizations can be made.

If the type of a variable is not declared, we can infer that the variable has a more specific type in two circumstances. First, when all settings of the variable return a common type more specific than the declared type, then the variable may be declared to always contain the more specific type. For example, if a variable is bound and initialized to the value of a +f, we may then infer that the variable will always hold a fixnum, which is further optimized to Sint32. Second, when references that require a more specific type unconditionally follow every setting of the variable, then it will be an error for the variable to have a type other than the more specific required type, and so we may infer that the variable may be declared to have that type, pushing the type requirement of the references further "upstream" towards the setting expressions. For example, this often happens in

functions that receive frames as arguments. The slot accessors expand into svref calls, which require simple-vector arguments so the function argument holding the frame may be optimized to simple-vector, and further then to Obj*.

Note that neither of these optimizations should be taken for variables that have explicit type declarations. The translation should consider the types of those variables to be fixed. If some dependency is being made on a function argument to be type Obj in the translation, then a user could type declare that argument to be type T, turning off all attempts to specialize that argument.

3 Memory Architecture

Use pointers to a one word (4 byte) header structure as the basic mechanism. The type should be an 8-bit unsigned byte value. The upper 3 bytes of the header word should be used for type specific data. For vectors this should be the length, giving us a simple vector with one word of overhead (all previous Lisp implementations we've used, except the Lispm, had two or more words of header). All Lisp objects should be aligned on 4 byte addresses. There is an argument that they should be aligned on 8 byte addresses. There are two advantages to 8 byte alignment. One is that it would enable us to reliably store immediate double floats and pointers into the same array. The second is that it would can give us an immediate type tag for conses that is itself the offset to the cdr of a cons. The argument for 4 byte alignment is that all current C compilers we use align structures on 4 byte addresses, but some won't align them on 8 byte addresses, even if they contain doubles. The other argument for 4 byte alignment is that we would not need to occasionally skip forward 4 bytes when allocating from heap in order to find the next 8 byte aligned address. For now, the 4 byte alignment wins.

The exceptions to the pointer to a type tag rule are the following types, which have immediate type tags. These types are fixnums, conses, and managed-floats. If all pointers are 4 byte aligned, then the lower 2 bits of all pointers are always zeros. This gives us 3 non-zero immediate type tags. The following are the immediate type tag assignments:

```
0:  pointer to header of Lisp object
1:  immediate fixnum
2:  pointer to cons (mask with -4 for car*, follow for cdr*)
3:  immediate character
```

A determination of type can be made from a hex value of the pointer.

```
hex 0, 4, 8, C are pointers to Lisp objects
hex 1, 5, 9, and D are fixnums,
hex 2, 6, A, and E are pointers to conses
hex 3, 7, B, and F are immediate characters.
```

The arguments for a 30 bit fixnum instead of a 31 is that it makes for fast fixnum additions and subtractions without risking overflow. 31 bit fixnums require a sequence point between operations to avoid the potential for overflow.

Characters could be either immediate or remote, though there is some performance benefit to characters being immediate. We could have a preallocated array of remote character objects, and allocate a Lisp character is arefing into this array. For now we are going with immediate characters, especially since we are thinking hard about UNICODE characters, which require 16 bits and so make a pre-allocated array become somewhat too large.

The type tags in headers should not conflict with any of the immediate type tags, so that type-case can turn into a fixnum-case of the type values.

Given these set ups, type tests against straight types are at worst a null test, a mask and an integer equality test, a character byte fetch, and another integer equality test.

In C, the type for object should be an unsigned integer type 32 bits long. There are several advantages to this. All architectures (including the Alpha OSF when using a linker

option) can have pointer values be represented in 32 bits, but for some platforms (the Alpha OSF) pointer types can consume something other that 4 bytes, 8 for all pointers on the Alpha OSF. By forcing all objects to consume 4 bytes, we can get interesting packing in structures and vectors on all platforms. [The following comment applies only to 8 byte alignment: For example, we could put immediate floats into odd-indexed simple vector locations, and those floats would consume 2 elements. For structures and frames, this could provide significant savings. -jra 8/30/95]

The type for managed float should have an immediate type tag so that we may have the smallest possible representation for floats, since so many are used. For each type that involves a heap allocated block of memory (i.e. all but fixnum and characters), there should be a corresponding type for a reclaimed instance of that type. This gives us a fast means of testing if a data structure is currently reclaimed or not, and would cause type tests in a safe translation. Even in production systems, we could check for double reclamation of data structures. Note that this is not possible for heap allocated data structures that use an immediate type tag.

The type tags for heap allocated data structures have no special issues, except that these values should not collide with any immediate type tags (i.e. be greater than 3), and be able to quickly determine if a data structure is reclaimed (i.e. use bit 7 as a flag).

So the type tag table is as follows:

```
Tag Value Reclaimed Tag
Dec     Hex Lisp Type C Type & Conv Dec    Hex
------------ --------- ------------ -------------
0 00 immed pointer (Header *)Obj
1 01 immed fixnum ((sint32)Obj)>>2
2 02 cons (Obj *)(Obj-2)
3 03 immed character (unsigned char)(Obj>>2)
4 04 managed-float (Mdouble *)Obj 132 84
5 05 double-float (Ldouble *)Obj 133 85
6 06 simple-vector (Sv *)Obj   134 86
7 07 string (w/fill ptr) (Str *)Obj   135 87
8 08 (simple-array ubyte 8) (Sa_uint8 *)Obj   136 88
9 09 (simple-array ubyte 16) (Sa_uint16 *)Obj  137 89
10 0A (simple-array double) (Sa_double *)Obj  138 8A
11 0B symbol (Sym *)Obj   139 8B
12 0C compiled-function (Func *)Obj   140 8C
13 0D package (Package *)Obj   141 8D
```

Each of the data structures are described below.

3.1 Simple-vectors

Simple vectors are the most often used data structure, so we'd like to keep it as small and fast as possible. It has a one word (i.e. 4 byte) header. The only components of a simple vector are the type tag, the length, and the body of the array. If the type tag is 8 bits wide (unsigned) then the length can be 24 bits of unsigned integer, giving a maximum length of 16 Meg.

```
typedef struct {
  unsigned int:8:type;
  unsigned int:24:length;
  Obj[1]:body;
} Sv
```

In C references to array are guaranteed to not be bounds clicked. this means that this one type can be used to all elements of arbitrarily sized simple vectors. Constant vectors can be made by having a type of simple-vector local to the C file containing the constant so that we can use an initialized structure. For example, a constant simple vector containing 5 fixnums could be emitted as C code as follows.

```
typedef struct {
  unsigned int:8:type;
  unsigned int:24:length;
  Obj[5]:body;
} Sv_5

static Sa_5 const1 = {8, 5, {fix(23), fix(2), fix(3),
  fix(9), fix(-12)}};
```

An example of the translation of a fetch of a simple vector element follows.

```
  Lisp: (setq x (svref y 5))
  C:    x = (Sv *)y->body[5];
```

Though the detail of casting and fetching the body component could be hidden in a C macro, at first we will leave all the details exploded out. One criticism of Chestnut's translations is that no one can figure out what the implementation is actually doing. Exploding out the details will help train development in the details.

3.2 Strings

In order to give fast performance for all strings, we will have fill pointers for all strings in G2. The fill pointer and length will both consume 3 bytes, plus one byte for the tag. The body of the string will be packed directly against the fill pointer, meaning that the most efficient packing of strings into our 8 byte aligned space will be for strings with lengths that have mod (length, 8)=1. Since strings in C must be null terminated, that adds 1 extra byte to the length as compared to the loop length of the string. Having 1 extra byte in the header word will allow us to efficiently pack strings into words when they have lengths that are multiples of 4.

```
typedef struct {
  unsigned int: 8: type;
  unsigned int:24: length;
  unsigned int:24:fill_length;
  char[9]: body
} Str
```

3.3 Immediate integer and double arrays

All of these types are implemented in the same way, with a 4 byte-aligned one byte type tag, a 3 byte length, and then a body. These types are Sa_uint8 and Sa_uint16. Sa_double is similar except that it is 8 byte aligned.. The type for bit-vectors is built on top of Sa_uint8, by fetching bytes and bit-shifting to fetch values and modify values.

3.4 Symbol

There are several different optimizations that one can imagine for the slots of symbols. In the first pass of this implementation, no such memory squeezing will be attempted, and all five typical slots of symbols will be directly provided in the symbol structure. Also, since virtually all symbols are included in packages in G2 and TW, the slots needed for holding the symbol in package balanced binary trees will be included in the symbol itself.

```
typedef struct {
  unsigned int:8:type;
  unsigned int:1:local_value;
  unsigned int:1:external;
  signed int:4:balance;
  unsigned int:1:imported;
  unsigned int:16:name_hash;
  Obj:symbol_name;
  Obj:symbol_value;
  Obj:symbol_plist;
  Obj:symbol_package;
  Obj:symbol_function;
  Obj:package_left_branch;
  Obj:package_right_branch;
} Symbol
```

The symbol-value slot is typically a pointer to the location containing the true symbol value, which will typically be a C global variable. When a runtime generated symbol has a value, the local-value bit will be 1, and the value is directly within the symbol-value slot. Note that slows down explicit calls to Lisp symbol-value since it has to check the local_value bit, but these explicit calls are rare and so I don't think that's so bad.

If packages are implemented as balanced binary trees, using the hash value of the name as an index, then there can be collisions between these hash values. In this case, the symbol names will be alphabetically ordered to determine left, right or match. These choices give us a constant size for symbols, given the size of the symbol-name.

3.5 Compiled Functions

Compiled functions will contain a pointer to the C function for this compiled function, the number of arguments for the function; the number of those arguments that are optional in the Lisp function, and a list of the constants that are default values for the optional arguments. Within the C runtime system, all functions will receive all arguments. If a funcall of a compiled function occurs where some optional arguments are not provided, the

default values are extracted from the list of default values. (In circumstances where the default value semantics require more than a constant, the no-arg argument value will be passed, and further computation will happen within the function to implement the default value selection).

```
typedef struct {
  unsigned int:8:type;
  unsigned int:8:arg_count;
  unsigned int:8:optional_arguments;
  Obj:default_arguments;
  void *f(): c_function;
} Func
```

The type spec for compiled functions will typically have arguments of type Obj: and a return type Obj: In special cases we may provide unsafe optimizations for functions that do not have object type arg and return values, but these should never be given to the default funcall operations.

3.6 Characters

Since all immediate type tags are in use, we will use heap allocated character objects. To avoid the need to reclaim them, we will preallocate all 256 character objects (at 8 bytes per, that's 2K). The value is explicitly unsigned to avoid confusion in parts that have a default signed char type.

```
typedef struct {
  double [0]: align;
  unsigned int:8:type;
  unsigned char:value
} Lchar
```

3.7 Package

Packages are implemented as balanced binary trees. They contain a list of used packages, and a symbol (or null pointer) that is the root node of the balanced binary tree. The balanced binary tree of symbol structures will hold only symbols that are local to this package and that have been imported to this package. External symbols of a package are represented by mask bits on the symbol itself. Imported symbols are represented by a new symbol structure in the binary tree, and its symbol value points to the imported symbol. The intern function will be implemented by searching through the binary trees of this package then through the binary trees of used packages and their used packages, but only accepting external symbols.

```
typedef struct {
  unsigned int:8:type;
  Obj:root_symbol;
  Obj:used_package_list;
} Pkg
```

3.8 Obsolete 8 Byte Alignments

[The following section describes tagging for 8 byte alignments. Keeping for historical perspective. -jra 8/30/95]

The exceptions to the pointer to a type tag rule are the following types, which have immediate type tags. These types are fixnums, characters, managed-floats, and conses. If all pointers are 8 byte aligned, then the lower 3 bits of all pointers are always zeros. This gives us 7 non-zero immediate type tags. The following are the immediate type tag assignments:

```
0:  pointer to header of Lisp object
1:  immediate even fixnum
2:  immediate character
3:  pointer to managed-float (mask with -8 for double*)
4:  pointer to cons (mask with -8 for car*, follow for cdr*)
5:  immediate odd fixnum
6:  pointer to double-float (mask with -8 for double*)
7:  unused
```

With this implementation, some determination of type can be made from the printed hex value of the pointer:

```
hex 0 and 8 are pointers to Lisp objects,
hex 1, 5, 9, and D are fixnums,
hex 2 and A are characters,
hex 3 and B are managed-floats,
hex 4 and C are conses,
hex 6 and E are double-floats,
hex 7 and F are unused, so must be corrupted objects
```

By having a cons tag of 4 and a header-less cons, the access to one of the elements of the cons will be a straight pointer dereference. The other requires subtracting 4 from the pointer first. (On the Alpha OSF or other 8 byte pointer, machines, some address computation is necessary for both the car and cdr, unless we make object a 4 byte wide data type, which we could do).

The type tags for heap allocated data structures have no special issues, except that these values should not collide with any immediate type tags (i.e. be greater than 7), and be able to quickly determine if a data structure is reclaimed (i.e. use bit 7 as a flag).

So the type tag table is as follows:

```
Tag (Hex) Lisp Type C Type  Reclaimed Tag
------------ --------- ------  -------------
0 00 immed pointer (Header *)Obj
1 01 immed even fixnum (sint32)Obj>>2
2 02 immed character (char)(Obj>>2)
3 03 managed-float (double *)Obj
4 04 cons (Obj *)Obj-1
5 05 immed odd fixnum (sint32)Obj>>2
6 06 double-float (double *)Obj
7 07 unused
8 08 simple-vector (Sv *)Obj  136 88
9 09 string (w/fill ptr) (Str *)Obj  137 89
10 0A (simple-array ubyte 8) (Sa_uint8 *)Obj  138 8A
11 0B (simple-array ubyte 16) (Sa_uint16 *)Obj 139 8B
12 0C (simple-array double) (Sa_double *)Obj 140 8C
13 0D symbol (Sym *)Obj  141 8D
14 0E compiled-function (Func *)Obj  142 8E
15 0F package (Package *)Obj  143 8F
```

Note the use of a zero length double array at the beginning of the struct. This ensures double word alignment, which gives us two things. The first is that pointers to constant structures of this type will be on 8 byte boundaries, required for our use of 3 bits of immediate type tag. The second benefit is that with a known alignment we can reliably mix objects and immediate doubles in simple-vectors. This is needed for structure and frame extensions I would like to make. The Harbison and Steele manual suggests that this zero length double array technique will ensure alignment, but this has not yet been tested.

4 Original Introduction

This introduction was written at the beginning of the effort to make a new Lisp to C translator. At the time I was working for Gensym Corporation, and named the project after the company. -jallard 10/9/99

The Gensym Language Translator (GLT) is being made to enable Gensym to control the means of distribution of its products. Since 1986 till now (January, 1995), we have used 6 different brands of Lisp compiler, abandoning each one in turn for platform availability, portability, and performance reasons. Most recently we have been using the Chestnut Lisp to C translator to deliver our Lisp-based products for releases 3 and 4. Between these two releases of our products, we took a new release from Chestnut, and it took a full man-year of my time to resolve the problems that had been introduced. I've made the judgment that for less labor than is devoted to maintaining our use of Chestnut's product, we can implement and maintain our own translator. It's my intent to implement a translator that meets or beats Chestnut's characteristics for the portions of Lisp that we use, and that does not waste time or attention on the portions that we do not use. Time can be taken later to extend it into areas we would like to use, but currently cannot because we can't depend on non-consing implementations.

The base of GLT consists of a small hand-written C include file, a small hand-written memory allocation C file, and a translator written in Common Lisp that can translate a minimal set of primitives from Lisp to C. The primitives in the translator implement data structure allocation and manipulation, the defining of constants, parameters, variables, macros, and functions. The system will not contain many of the features normally present in a full Common Lisp implementation, most notably no garbage collector, lexical closures, rest arguments, condition system, bignums, complex numbers, or CLOS.[1] The built-in types will be limited to packages, symbols, conses, fixnums, doubles, simple-vectors, simple-arrays of unsigned-byte 1, 8, and 16 bits, simple-arrays of doubles, and fill-pointered non-adjustable strings. Notable complex features that will be included are special variables, catch and throw, unwind-protect, multiple-values, optional arguments, defstruct, macrolet, labels, flet, and deftype.

Particular attention will be paid to using C integer and double types for variables, function arguments, and function values when type declarations allow, largely because it bugs me when these things aren't handled right and I believe it is a significant performance loss to always be boxing and unboxing at function boundaries. We will also make special efforts to optimize calls to functions that return a single value.

[1] Rest arguments have since been added. -jallard 10/9/99.

Index

R