Implementing Primitive Datatypes
for Higher-Level Languages

Stanley T. Shebs

Tech. Report. UUCS-88-020

TECHNICAL REPORT

Department of Computer Science

University of Utah
Salt Lake City, Utah

Copyright © Stanley T. Shebs 1988

All Rights Reserved

ABSTRACT

Implementation of modern programming languages is a complex task. Bridging the semantic gap between abstract linguistic constructs and concrete hardware components requires much software, including compilers, interpreters, runtime libraries, and programming environments. Compiler construction has long been aided by parser generators and attribute grammar evaluators, but the other components have been neglected, even though they constitute the largest parts of implementations of Lisp, Prolog, Smalltalk, and similar languages. Within an implementation, the representation of primitive datatypes such as numbers, lists, strings, and symbols require some of the most difficult decisions by the implementor. The effectiveness of type discrimination schemes, interactions between storage allocation and virtual memory, and general time/space tradeoffs are issues that have no simple resolution; they must be evaluated for each implementation.

The problems are approached from three directions: a survey of representation ideas used in existing systems, a set of design rules that mimic the behavior of expert implementors, and an automatic designer that generates the primitive datatypes of a Common Lisp system. This Common Lisp system can then be used to compare different representations in an accurate and reproducible manner. Transformations of one abstract type into another turn out to be important design steps, and the separation of the notion of function type into signature and contextual types is an essential part of code generation. Experimental results indicate that although the process can be made to work, completely automated construction of high-quality designs requires further advances.

To my parents

CONTENTS

ABSTRACT ii

LIST OF TABLES ix

LIST OF FIGURES x

ACKNOWLEDGMENTS xii

CHAPTERS

1. INTRODUCTION 1
   - 1.1 Many Languages, Many Implementations 1
   - 1.2 The Structure of Language Implementations 2
     - 1.2.1 Theory 3
     - 1.2.2 Practice 4
     - 1.2.3 Runtime Systems 5
   - 1.3 An Approach to Designing Data Structures 6
   - 1.4 Related Work 8
     - 1.4.1 Computer-Aided Implementation 8
     - 1.4.2 Data Structure Design 8
     - 1.4.3 Studying Tradeoffs 10
   - 1.5 How This Work Relates to Their Work 11
   - 1.6 The Rest of the Dissertation 12

2. REVIEW OF DATA STRUCTURE DESIGNS 13
   - 2.1 Lisp 14
     - 2.1.1 LISP 1 14
     - 2.1.2 7090 LISP 1.5 14
     - 2.1.3 M-460 LISP 15
     - 2.1.4 Q-32 LISP 17
     - 2.1.5 PDP-1 LISP 18
     - 2.1.6 LISP 1.6 and UCI Lisp 18
     - 2.1.7 LISP 2 19
     - 2.1.8 UT LISP 21
     - 2.1.9 BBN LISP 21
     - 2.1.10 1108 LISP 22
     - 2.1.11 LISP F3/F4 22
     - 2.1.12 MicroLISP 23
     - 2.1.13 PDP-10 MacLISP 23
     - 2.1.14 Multics MacLISP 24

2.1.15 Interlisp-10 25
2.1.16 LISP-11 26
2.1.17 ULISP 26
2.1.18 Cambridge LISP 27
2.1.19 CLisp 27
2.1.20 ByteLisp 28
2.1.21 Interlisp-VAX 29
2.1.22 Interlisp-D 29
2.1.23 Zetalisp/Symbolics 3600 30
2.1.24 Scheme Chips 30
2.1.25 NIL 30
2.1.26 FLISP 32
2.1.27 Franz Lisp 33
2.1.28 Portable Standard Lisp 33
2.1.29 FLATS 34
2.1.30 LeLisp 34
2.1.31 Tandem Lisp 34
2.1.32 T 35
2.1.33 Spice Lisp 35
2.1.34 Data General Common Lisp 36
2.1.35 S-1 Lisp 38
2.1.36 Kyoto Common Lisp 38
2.1.37 HP Common LISP 38
2.1.38 Extended Common Lisp 39
2.1.39 Lucid Lisp 39
2.1.40 Lisp/370 and Lisp/VM 41
2.1.41 PC Scheme 41
2.1.42 mini-Scheme 42
2.1.43 XLISP 42
2.1.44 VT-LISP 43
2.1.45 CScheme 43
2.1.46 GNU Emacs Lisp 43
2.1.47 SPUR Lisp 44
2.1.48 LMI K-processor 45
2.1.49 UtiLisp 45
2.1.50 A-Lisp 46
2.1.51 SIOD 46
2.2 Purely Functional Languages 47
2.2.1 IDRIL 47
2.2.2 Illinois FP 48
2.3 Prolog 48
2.3.1 C-Prolog 48
2.3.2 SB-Prolog 49
2.4 Object-Oriented Languages 49

2.4.1 CLU ... 49
2.4.2 Xerox Smalltalk-80 ... 50
2.4.3 Tektronix 4406 Smalltalk ... 51
2.4.4 Swamp ... 52
2.4.5 ConcurrentSmalltalk ... 52
2.4.6 Little Smalltalk ... 52
2.4.7 BrouHaHa ... 53
2.4.8 UMass Smalltalk ... 54
2.5 SNOBOL4 ... 54
2.5.1 The Macro Implementation ... 54
2.6 Icon ... 55
2.7 APL ... 56
2.7.1 Purdue/Unix APL ... 56
2.7.2 Q'Nial ... 57
2.8 Conventional Languages ... 57
2.8.1 Ada ... 59
2.9 Summary ... 59

3. DESCRIPTIVE FORMALISMS ... 61
3.1 Formal Definition of Types ... 61
3.1.1 Standard Schemas ... 63
3.1.2 Booleans ... 63
3.1.3 Integers ... 63
3.1.4 Ranges ... 64
3.1.5 Sums ... 64
3.1.6 Structures ... 65
3.1.7 Vectors ... 66
3.1.8 User-Defined Types ... 66
3.1.9 Examples ... 67
3.2 Formal Definition of Machines ... 69
3.2.1 Examples ... 70
3.3 Representation of an Implementation ... 70
3.4 Pragmatics of Usage ... 71
3.4.1 Finiteness ... 71
3.4.2 Statistical Patterns ... 72
3.5 Pragmatic Limits in Standardized Languages ... 75
3.6 Summary ... 76

4. IMPLEMENTATION DESIGN RULES ... 77
4.1 Global Design Issues ... 77
4.1.1 Feasibility ... 78
4.2 Machine Characteristics ... 78
4.2.1 Sizes of Objects ... 79
4.2.2 Object Tables ... 79
4.3 Design for Integers ... 80

4.4 Design for Sums 82
4.4.1 Design of Tags 82
4.4.2 Design of Separate Spaces 85
4.4.3 Design of BBOP 86
4.4.4 Combinations of Sums 87
4.4.5 Merging and Splitting Sums 87
4.5 Design for Structures 89
4.6 Design of Vectors 91
4.6.1 Vector/Structure Integration 94
4.6.2 Arrays 94
4.7 Other Types 95
4.7.1 Floating Point Numbers 95
4.7.2 Rational Numbers 97
4.8 Special Considerations 97
4.8.1 List Compaction 97
4.8.2 Storage Reclamation 97
4.9 Summary 101

5. AN AUTOMATIC DESIGNER 103
5.1 The Designer 104
5.1.1 Making Decisions 105
5.1.2 Choice Objects 105
5.1.3 Type-to-Type Transformation 106
5.1.4 Finishing the Design 108
5.2 The Coder 110
5.2.1 Division and Flattening 111
5.2.2 Instruction Matching 111
5.2.3 Generating Files 111
5.3 Utah Common Lisp 113
5.3.1 Itemization 114
5.3.2 Code Generation 114
5.3.3 Register Allocation 116
5.3.4 Assembly 116
5.3.5 Micro-kernel 116
5.4 Evaluation 117
5.4.1 Benchmarks 117
5.4.2 Results 117
5.5 Discussion 119
5.5.1 Lack of Generality 120
5.5.2 Combinatorial Explosion 120
5.5.3 Coding Optimizations 121
5.5.4 Use of Registers 122
5.6 Summary 122

6. CONCLUSION . . . 123
6.1 Contributions . . . 123
6.1.1 Recommendations for Lisp Standards . . . 124
6.2 Extensions . . . 125
6.3 Applications . . . 126
6.4 Abstract Data Types in General . . . 127

APPENDICES

A. SPECIFICATIONS OF TYPES . . . 129
A.1 Basic Lisp Definition . . . 129
A.2 Common Lisp Definition . . . 129
A.3 68000 Definition . . . 132

B. COMPLETE DESIGNER RUN . . . 135
B.1 Designer Session . . . 135
B.2 Abstract Design 0 . . . 138
B.3 Opencodings for Design 0 . . . 139
B.4 Primitives for Design 0 . . . 140
B.5 Bootstrap File . . . 141
B.6 Benchmark Run . . . 141

REFERENCES . . . 142

LIST OF TABLES

2.1 Tag and Data Sizes of Various PSL Implementations 34

5.1 Designs Produced 119

5.2 Execution Times 119

LIST OF FIGURES

1.1 Structure of a Common Lisp Implementation 6

2.1 Fields in IBM 704 Word 14

2.2 Memory Allocation in LISP 1.5 16

2.3 The Symbol CHARCOUNT in LISP 1.5 17

2.4 Number Representation in LISP 1.6 19

2.5 Data Structures in LISP 2 20

2.6 UT LISP Word Layout 21

2.7 Data Representation in PDP-10 MacLISP 24

2.8 A Multics Maclisp Pointer 25

2.9 Data Representation in LISP-11 27

2.10 Tag Assignment in Cambridge LISP 28

2.11 Data Representation in Zetalisp 31

2.12 Data Representation in NIL 32

2.13 Use of Low Tags in T3 36

2.14 Data Representation in Spice Lisp 37

2.15 Representation of NIL in ExCL 40

2.16 A Data Object in SPUR 44

2.17 Data Representation in C-Prolog 49

2.18 Objects in Smalltalk-80 51

2.19 Object Header in 4406 Smalltalk 52

2.20 Objects in Little Smalltalk 53

2.21 Representation in SNOBOL4 55

2.22 Representation in Icon Version 6.2 57

2.23 Q'Nial Array Representation 58

3.1 An Implementation Function 71

3.2 Symbol Name Lengths in HP Common Lisp 74

4.1 Graph of BBOP Tradeoffs 87

4.2 Allowable Combinations for Type Discrimination 88

4.3 IEEE Floating Point Format 96

5.1 Tag Assignment Choice 106

5.2 Generator of Tag Assignment Code 107

5.3 A Type-to-Type Transformation 109

5.4 Addition of Wrappers to a Primitive 110

5.5 Machine-Generated Opencodings 112

5.6 Machine-Generated Primitives 113

5.7 Architecture of the UCL Compiler 115

5.8 Evaluation Process 118

ACKNOWLEDGMENTS

Bob Kessler deserves the most credit for being supportive while I chased wild ideas around. He has made the PASS group a very enjoyable place to be over the past five years. Gary Lindstrom was instrumental in supplying doses of reality seasoned with humor. He and Bob Keller suffered me to talk about almost anything in the AMPS seminar. Ganesh Gopalakrishnan agreed to be on my committee almost at the last minute, and made many useful remarks on the draft. John C. Peterson and Julian Padget offered good advice, while Gerry Sussman said to read a lot.

Jed Krohnfeldt and Harold Carr have been good colleagues, even though it seems like we're always working on completely different things at any one time. Leigh Stoller deserves a hand for having faith in the success of the UCL system, and spending long hours on it. Other members of the PASS group endured inchoate talks and formless ideas: Jerry Duggan, Jeff Knell, Will Galway, Mike Hucka, Bobbie Othmer, Eric Muehle, Mohammad Pourheidari, and Craig Steury. John Brasher was an endless source of thoughts and conversation. Ted Jardine gave me an opportunity to do some of my research at Boeing, and protected me from management. The two John Petersons (John C. and John W.), Marti Peterson, Lal George, Tim Moore, and John van Rosendale helped keep me sane by wanting to go rock-climbing even when I thought I was too busy. Sandra Loosemore has been both a special friend and an valuable colleague; few women can match her accomplishments.

Many people have helped with the survey in Chapter 2, most notably Eric Benson, Fons Botman, George Carrette, Dan Corkill, John Cowan, Jeff Dalton, John Fitch, John Foderaro, Bernie Greenberg, Joe Marshall, Eliot Moss, Dave Moon, Joe Murray, Eric Norman, Per-Eric Olsson, Paul Pedersen, Jonathan Rees, Bob Shaw, JonL White and Kei Yuasa. I hope I have not misrepresented their systems too seriously!

The Amoco Foundation was very important in the gestation of this work. Their three years of funding gave me time to learn and to develop ideas freely. My activities in the last two years were partly supported by the Hewlett-Packard Company and by DARPA under contract number DAAK11-84-K-0017. Finally, the Boeing AI Center supported an early version of this research during the summer of 1985.

CHAPTER 1

INTRODUCTION

> If controversies were to arise, there would be no more need for disputation between two philosophers than between two accountants. For it would suffice to take their pencils in their hands, to sit down to their slates, and to say to each other (with a friend as witness, if they liked): *Calculemus* [Let us calculate].
>
> G.W. Leibniz (*ca* 1670)

High-level programming languages offer the opportunity to improve computing practice, by abstracting away from hardware, and towards useful applications. Lisp, Prolog, Smalltalk, APL, SETL, Snobol, and Icon, to name some of the better-known languages, include among their abstractions a variety of predefined data objects, including lists, symbols, sets, and character strings. However, builders of these languages have always faced a series of puzzling questions about the representation of data objects. How should the type of an object be recorded? What is the best memory layout for complex objects? Will multiple representations offer any space or speed advantages? How should memory be allocated and reclaimed? To help answer these questions, this dissertation will introduce the systematic investigation of data object implementation, both by the study of existing systems, and by the construction of an automated designer of implementations.

1.1 Many Languages, Many Implementations

In the 1930s and 40s, the pioneers of computing anticipated many ideas; but they never dreamed of the modern proliferation of programming languages. In 1966, Landin mentioned a survey that had counted 1,700 languages [91], although Sammet's stricter criteria counted a mere (!) 120 in her famous book [133]. No one even bothers to try counting them today. Feelings are mixed on whether this diversity is good or bad, but in any case, the abundance of languages seems to be an enduring feature of the computing scene.

Diversity creates at least one indisputable problem. Each and every language in actual use must have one or more programs which *implement* the language. Language implementations are crucial pieces of software, comparable to text editors and operating systems in their importance for computing. Customers accept or reject

computer systems on the basis of the available languages; the more sophisticated users will also look at the performance of the languages, since poor software can bog down a system tremendously.

Language implementations are more like operating systems than text editors, however; although language systems can be made portable, efficient ones are invariably hardware-specific. Combined with the multiplicity of languages, this fact leads to a situation in which a typical computing center has dozens or even hundreds of language systems in active use. Worse, each such system is a medium- to large-sized program or collection of programs.

Given this, it should not be too surprising that efforts have been made to regularize and automate the construction of language implementations. Since languages were originally equated with translators and compilers, work has largely concentrated on the automation of compiler writing. Great successes have been achieved in this area; automatic parser generation has reduced the syntax analysis phase from a mystery to a few days' programming task, while the use of common intermediate languages has increased the sharing and reuse of compiler components. The success of this process can be seen by comparing the 18 person-years needed to construct the first Fortran compiler [11] to the one or two person-years typically allotted for compilers of similar quality nowadays [4].

The times have changed in other ways as well. Interest has been slowly but steadily growing in "more abstract" or "higher-level" languages: primarily, but not exclusively, languages whose basic model of computation is not based on the conventional von Neumann machine. The models of computation vary widely, from function application (Lisp, Logo, ML), to logic (Prolog), to message passing (Smalltalk, Actors), to string/pattern matching (Snobol, Icon), to array operations (APL), to set theory (SETL). Also, the programming environment has become of central importance, even for conventional languages. A modern environment is not complete without source-level debuggers, formatting tools, and language-sensitive editors. Compiler construction alone is no longer sufficient.

In many ways, the present situation for new languages and environments is much like it used to be for compilers: relatively few were built, and each one was as likely to be a research effort as a programming project. Unfortunately, the paucity of publications on the construction of these systems has led to a situation in which many ideas have been reinvented or ignored.

1.2 The Structure of Language Implementations

Although the general theory of language implementation has been studied somewhat, much of the practice is still rather mysterious. It is mostly unpublished; concepts and techniques are passed on by word-of-mouth, by comments in source code, or by cryptic remarks in the backs of manuals. Thus, a brief sketch of theory and practice is appropriate. For a more extensive view of practice, see Lecarme and Gart's book on software portability [92, ch. 5].

1.2.1 Theory

Semantic theory has relatively little to say about the necessary structure of a language system. If we use a denotational formalism such as that described in an introductory text [152], every language is a member of the class of functions mapping *programs* to *behaviors* (which are themselves mappings from inputs to outputs):

$$L \in (P \to (I \to O))$$

Programs are objects in what are typically called *syntactic* domains, although such domains have no truly unique characteristics. A program could be a string of characters, a sequence of tokens, a tree-structured object, or something else entirely. The class of behaviors is similarly unconstrained, since "behavior" could range from a constant function that returns the same thing no matter what the input, to the behavior of a spreadsheet program, to the complexities of human activity. There are no other restrictions; this means that for any given program and any given behavior, there exists at least one language that causes the given program to exhibit the given behavior. In other words, the class of possible languages is extremely large.

Note that there is no inherent notion of compilation in this model. The function L is always an "interpreter." Compilers arise as a *frontend* to the interpreter, by translating the source language into another language which is itself interpreted in order to get I/O behavior:

$$L = L' \circ C$$

where

$$C \in (P \to P')$$
$$L' \in (P' \to (I \to O))$$

C compiles an L program in P to a program in P'. L' may range over a variety of languages, from a slightly altered version of L to raw machine language. In the latter case, the "syntactic" domain P' consists of sequences of machine language instructions.

The process of dividing a language into a compiler and another language may be repeated indefinitely, yielding a sequence of compilers each feeding the next in line. This view leads to some useful observations; for instance, the interpreter of a Lisp system operates on S-expressions and not the source text, which means that the Lisp reader can be considered to be a simple compiler, and that compiler techniques may be applicable. Likewise, intermediate languages in a compiler could have interpreters corresponding to them. An attempt to write such an interpreter is a useful way to detect omissions in the intermediate language and/or mistakes in code generation and optimization.

The basic fact derived here can be simply stated:

Compilers and interpreters always come in pairs.

The significance of this is that it governs the language implementation process, whether or not implementors are consciously aware of it.

1.2.2 Practice

The practice of language implementation centers around the basic fact of the last section; how many compiler/interpreter pairs should be defined, and what should be the division of labor for each pair? At one end of the spectrum, a C compiler does almost everything, and has an "interpreter" consisting of the hardware augmented with a few hundred bytes of runtime system code. At the other end, a modern Lisp system may include several types of compilation, multiple optimization options, and several megabytes of runtime system. These decisions define the basic architecture of an implementation, and are among the first choices made by an implementor.[1]

There are very few purely interpretive systems. To qualify, a system must interpret source code directly, which usually means looking at strings of characters. Some early microcomputer implementations of BASIC did work this way, but the cost of repeated lexical analysis on symbols and numbers is quite high; for instance, a constant number inside a loop would have to be read and reconstructed on each iteration through the loop.

A more reasonable approach to interpretation employs a lexical analyzer and maybe a parser to produce an intermediate form with a fairly regular structure, such as a sequence of tokens, but still close enough to the source form that it could be reconstructed. The majority of interpreters work this way, and in some cases the intermediate form is formally described and of interest in its own right. (Lisp S-expressions and Prolog databases are two well-known examples.) Some implementations may do a considerable amount of preprocessing. Hewlett-Packard's Common Lisp [70] transforms S-expressions into another intermediate form in which lexical variables have been alpha-converted and macros have been expanded. Although the result is still used by something resembling a Lisp interpreter, it is very different from the source code, and reconstruction of the exact source code is not possible.

Next, an implementation may compile to an abstract machine language and interpret that language. Perhaps the best-known example is Pascal P-code, which has been implemented in hardware, but is far more often seen as input to an interpreter [14,120]. P-code is machine language for a stack machine, which simplifies the compiler greatly. Other examples include Scheme 84 [52] and PC Scheme [16], as well as Snobol4 [66]. Almost all Prolog compilers use the "Warren Abstract Machine" (WAM) designed by D.H.D. Warren [162]. The abstract machine technique is quite popular in research environments where new architectures are being proposed. The interpreters can be complicated or simple, depending on the complexity of the simulated machine and the level of detail required. Such a low-level language also appears in conventional compilers, but the only "interpreter" is the compiler backend and its target [4]. The abstract machine need not be similar to conventional hardware; popular lines of research include abstract machines based on dataflow [9] or graph reduction [122].

---

[1] It should be noted that these decisions may be made by external forces or by custom, so they are not always explicitly stated.

Finally, a language implementation may compile to real machine code. This approach is favored for commercial-quality implementations, where the order of magnitude performance advantage over other approaches justifies the order of magnitude increase in system size and complexity. Even in this situation of "total compilation," the compiler can rarely if ever generate raw machine instructions only. Instead, it will compile to what has been referred to as the "Compiler Writer's Virtual Machine" [30], a combination of hardware and low-level software.

Real implementations frequently use multiple pairs of compilers and interpreters. Many commercial compilers use several intermediate languages (each one forming pairs with the previous and following languages at each compiler stage), while typical Lisp implementations include read/eval and compiler/runtime pairs.

1.2.3 Runtime Systems

One notable characteristic of Lisp and Smalltalk systems is the tremendous size of the runtime system. Common Lisp, for example, defines some 600 functions [146], of which less than 100 are genuine primitives—the remaining functions are usually defined in terms of those primitives. The code for these definitions generally runs to about 40,000 lines of Common Lisp. The Smalltalk-80 Virtual Image contains hundreds of classes, each with 5-10 methods, and over 10,000 objects, all adding up to about 1/2 megabyte of data [90]. Much of this code is associated with window management, editing, and similar high-level operations. APLs and Prologs do not typically have such enormous runtime systems, although that may change in the case of Prolog; Quintus Prolog includes some 700 predicates [125].

Closer examination of nonprimitives reveals two partially overlapping classes of functionality: simple operations, and subsystems. Simple operations may range from set operations to elaborate sorting utilities, but even a very complicated sort function will still be simpler than a debugging package or an editor. In turn, the primitives are even simpler in their behavior.

Figure 1.1 summarizes this view of the various levels of a language system. Our chief interest in the succeeding chapters will be the level just above the machine—the primitive datatypes. This is where machine and language semantics interact directly, and where the most difficult design decisions arise.

Why are primitives difficult to implement? In terms of quantity of code, they are the smallest part of the runtime system, so optimizing their construction would seem to have little effect on the magnitude of the task. Nevertheless, there are several reasons to concentrate on primitive datatypes:

1. Machine dependencies. The primitives (by definition) cannot be expressed in terms of the language, and must be coded in some other way, usually involving machine language. By contrast, even though nonprimitives constitute greater volume, they are simpler to write, and may even be portable (the Spice Lisp library has been reused in many other Common Lisps, and the Smalltalk-80 Virtual Image has also been widely distributed [90]).

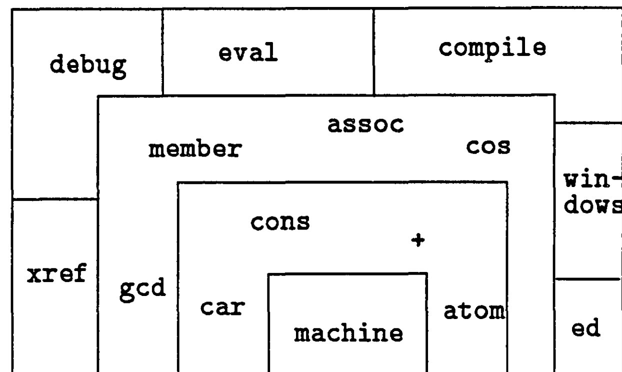

Figure 1.1. Structure of a Common Lisp Implementation

2. Operating system dependencies. OS facilities will be accessed through the primitives. Memory allocation may be constrained by the OS; most versions of Unix,[2] for example, are not configured to allow the use of an entire virtual address space.

3. Severe constraints on time and space. Primitives represent overhead relative to native machine structures, so poor performance will affect every program running in that system.

4. Complex tradeoffs. Some simulated results collected previously [141] clearly illustrate how even very simple designs for the primitives yield drastically different results for different benchmark programs.

Performance is by far the most important consideration, since primitives in higher-level languages can consume 50% or more of program execution time [137,150]. This is generally regarded as overhead relative to the use of lower-level languages (whether rightly or not is open to question), and is frequently used as a strong argument against higher-level languages. In response, human designers have emphasized the use of clever techniques to save a bit or a clock cycle here and there.

1.3 An Approach to Designing Data Structures

The goal, then, is to invent a method for implementing the primitives of a language. The method should be sufficiently precise that it could be incorporated into a program, if desired. Success will be measured according to the usefulness

---

[2]Unix is a trademark of AT&T.

of the results in real implementations, either directly, by generating part of an implementation, or indirectly, by helping a human designer make better decisions.

The first step is to reduce the problem to an exercise in the implementation of abstract data types (ADTs). Although the reduction is straightforward, the problem of ADT implementation is largely unsolved, so we will cut the Gordian knot and use heuristic rules in a generate-and-test paradigm. The rules create a number of plausible designs, and a subsequent coding stage produces definitions whose time and space costs can be estimated, or used directly in a real system.

This approach does *not* purport to come up with the "best" design. One of the results of this investigation is that the performance of designs varies so widely from language to language, machine to machine, and benchmark to benchmark, that any single design will turn out to be highly undesirable in many cases.

Let us look at an example of building a Common Lisp system for the 68000 processor. Common Lisp is a rather large language, so we shall consider only lists, small integers, characters, and strings. To build the implementation, we will need code for list functions (`cons`, `car`, `cdr`, `consp`), arithmetic operations (`+`, `-`, `>`), and string functions (`char`, `length`). The first step is to construct specifications for these types. The details are in chapter 3; for now it is sufficient to say that a list cell is a structure with two components (car and cdr), both small integers and characters are finite ranges of integers, and that a string is a varying-length vector of characters. We also need some limits, so we say that programs will use at most 100,000 list cells and up to 100,000 strings, each with no more than 10,000 characters, but that average string length is more like 80 characters, and that the total will be only 100,000 characters in all the strings together. The machine also needs a specification that says it can address 16 megabytes of memory, and that it has instructions for addition, subtraction, and so forth.

The design system is a Lisp program, and the specifications are named by symbols, so doing something like `(impl 'small-cl 'm68k)` sets it to work. The result will be a number of different designs, but for simplicity, let us consider only the rules involved in one design.

1. The toplevel part of the specification is a union of four types, and the rule defines two tag bits to be stored in the most significant bits of a word, with the tag value 0 assigned to lists, 1 assigned to small integers, and so forth.

2. The list specification is of a structure of two components *car* and *cdr*, and the rule matching this declares a structure in memory with the *car* stored at a lower address than the *cdr*.

3. Both the small integers and character specifications are ranges of integers smaller than a 32-bit word. The same rule matches each individually, and represents them one-to-one with twos-complement integers (the "natural" integer representation for a 68000).

4. The default rule for vector representation matches the string specification, which has the effect of representing the string as a block of words, where the

first word is a length, and each succeeding word contains a character object. Note that this is *not* the conventional byte-packed representation for strings.

Similar rule firings are occurring for other designs simultaneously, ultimately producing a large collection of different designs.

The later stages of the design system take the basic definitions of the primitive functions and transform them from an abstract language into machine code. The process is basically a mechanical one, very similar to standard compiler generation methods. The current directory now has a large number of files. The most interesting are the files `c`*n*`.l` and `p`*n*`.l`, which are sets of definitions used by the compiler and runtime system, respectively. The compiler definitions are for loading into a cross-compiler, which will then compile both the runtime system definitions and any user programs. Note that the correctness of the design is never proved explicitly, but is inherent in the correctness of individual transformations, which will be assumed.

1.4 Related Work

Apparently, completely automated construction of a programming language's primitive operations has never been done before. There has been a variety of work on similar problems, but under several different rubrics, including Data Design, Automatic Programming, Abstract Data Types, Program Transformation, and Very-High-Level Languages. Some work has also been done on the analysis of individual design tradeoffs for implementations.

1.4.1 Computer-Aided Implementation

The computer has long been used to aid compiler construction; in fact, the automatic generation of lexical analyzers and parsers is generally considered to be a solved problem [69]. Automatic construction of code generators has been a mixed success; Cattell [30], Fraser [50], Ganapathi [54], Graham and her students [56], R. Kessler [82], Pleban [123], and others have built working generators of code generators, but all have proved to have various defects and limitations [116], and have not (yet) supplanted custom-built code generators. Peephole optimizers have more recently proved amenable to automation, as demonstrated by Davidson and Fraser [37], and Kessler [83]; it remains to be seen whether they will be widely useful. There has been a little work on the automation of assembler construction [101,140,168]. Thus, although runtime systems have not been considered, successes in other areas give us reason to hope for success in automating runtime system construction.

1.4.2 Data Structure Design

What is perhaps the first work on data structure selection was done by Gotlieb and Tompa in the early 70s [59]. Their algorithm selected a representation from a

catalog of about a dozen possibilities, which encompassed representations ranging from linked lists to balanced trees. The datatype being implemented was a simple sort of database object that defined insertions, deletions, and lookups. The algorithm took as input the space available, the number of elements of each substructure, the size of the search key space, and so forth, and produced a shorter list of plausible representations. Each of these was subjected to a further evaluation that counted individual operations and produced a set of linear functions expressing the time cost of those operations. The best representation evaluated to the lowest final cost.

The 70s also saw extensive efforts in automatic programming as a problem in artificial intelligence (the *Handbook of AI*[13] has a survey chapter). Several projects featured data structure design as a key step.

Barstow's synthesizer PECOS [15] had sets of rules operating on the abstract types *collection* and *mapping*. The rules implemented the types as lists, bit arrays, hash tables, and several other Lisp object types. The rules emphasized finding valid implementations rather than optimizing them; it was intended that Kant's program LIBRA [80] would be the optimization expert. Barstow's rules were heuristic, since they were intended to reflect expertise in programming, and so were not capable of generating all possible implementations (although on at least one occasion they did produce an unanticipated design [15, p. 208]). LIBRA worked hand-in-hand with PECOS; its main responsibility was the estimation of costs of partially refined programs. Each data structure had cost formulas associated with its operations, and LIBRA used branch-and-bound to eliminate partially refined programs whose estimated costs were too high.

Low also did work on data structure selection for abstract sets and lists in a subset of SAIL [100], using a hill-climbing algorithm and user input to decide on the best representation, as well as program execution using simple default implementations to generate statistics. The representation library included eight ways to represent sets (sorted lists, bit vectors, trees, etc), and three ways to represent lists (singly and doubly linked lists, as well as varying length arrays). Low noted that the machine's attempts to infer usage of objects were not very successful, that user interrogation was generally necessary.

Rowe and Tonge [131] defined a somewhat elaborate mechanism for defining abstract datatypes, including capabilities to define distinguished elements of the type, various axioms such as commutativity of operations, and the kinds of operations supported by the type. The description of implementation types is similar. The two kinds of types are matched using a general algorithm. Although the method appears to have been effective, it was not completely implemented, and does not appear to have been pursued further.

More recently, interest in automatic programming has waned, having been replaced by activity in abstract data types and object-oriented programming. Such languages support free choice of representation quite well, but this support tends to be ignored, and all objects end up with the same representation, even when this is not appropriate. Mary Shaw called for multiple representations at a conference

in 1976 [136], but this call has gone largely unheeded. Notable exceptions are the languages SETL and Paragon.

Although SETL has been around for many years [134], it uses some modern ideas internally. SETL is based on the idea of sets as primitive objects, but each set may be implemented differently. The SETL compiler must manage operations involving different representations for sets, emitting code for coercions as necessary, and it uses flow analysis to decide which variables should get which representations so as to minimize the number of coercions. For further details on this, and some performance results, see [51]. The limitations of SETL are that it uses only a small and fixed set of implementations for sets, and that the language requires the representation of everything in terms of sets and mappings.

Paragon was developed by Mark Sherman, who described it in his dissertation [142]. It is basically an ADT language along the lines of Alphard or CLU, but offers *policy procedures* as a means for compiletime selection of representations. A policy procedure is a piece of Paragon code that is executed during compilation, and that decides which of several predefined representations will be used at any point. Although this is a powerful mechanism, it is totally manual, and in practice, most programmers would probably not take the time to code several representations, or to check that the policy procedures are really choosing the best representations.

Kapur and Srivas [81] have studied the use of a term rewriting system to implement one datatype in terms of another. The basic technique is to find theorems about representation instead of proving the correctness of given representations; to this end, Kapur and Srivas use rewriting to expand as well as reduce terms. The goal is to rewrite operations expressed in terms of basic operations, into operations expressed in terms of the implementation type. This very general approach was only applied to the implementation of a queue using a list, and key steps were done manually. Still, this paper shows some significant advantages in the formalization of individual transformation steps.

Jalote [74] has also investigated the automatic implementation of ADTs, but instead of using rewriting strategies, the axioms are classified into categories, each of which has a simple implementation. The overall representation is always the same; a tree. One significant point of this work is that the output is working C code. On the other hand, issues of consistency and completeness of axioms are ignored.

Darlington [36] has been an important exponent of the use of program transformation techniques to implement abstract data types. Most of the work seems to have concentrated on the application of transformations, rather than the automated discovery of useful transformations or "best targets" of transformations.

1.4.3 Studying Tradeoffs

Very recently, some interest has been developing in the tradeoffs inherent in language implementation. This has to some extent been spurred by the development of specialized hardware for languages, particularly Lisp. The choices of data structure representation become especially critical, since any mistakes will

be permanently enshrined in silicon. Because of this, most of the tradeoff studies have been conducted as part of a hardware design project. Steenkiste's dissertation [150] is particularly useful, although the concrete results are based on the Portable Standard Lisp implementation, which has many assumptions wired into it. Shaw's dissertation [137] also has extensive experimental results on HP Common Lisp, but only limited assessment of alternative implementations.

1.5 How This Work Relates to Their Work

In contrast to the previously-cited work, this dissertation will concentrate on data structures for existing high-level languages, primarily but not exclusively Lisp. This angle is at once more and less ambitious than previous work: more ambitious, because the goal is to generate "wizard-quality" designs to be used in real situations; less ambitious, because the data to be represented is simpler than those typically occurring in application programs. More specifically, primitive datatype design has several special characteristics:

- *Unlimited design time.* Primitive datatype design is basically a one-time process. This means that a variety of designs can be evaluated, even the less likely ones, and the evaluation can be done in a more realistic setting, perhaps even testing the implementation on the programs of special interest. The choice of test programs can greatly alter the apparent desirability of any given design. Designs might be redone occasionally, perhaps following a detailed performance study.

- *Numerous interacting design decisions.* Each design decision involves only a few alternatives, but there may be 20 to 30 such decisions made in a complete design. Good datatype designs tend to be tightly interwoven; there are many interacting considerations. Speed for one operation will be gained at the expense of another, while limited memory will have many competing demands placed on it. Representation "puns," where the same bits have multiple interpretations, are a staple of the better designs, and we will see some interesting examples in the next chapter.

- *Scarcity of experts.* Very few people have built even one higher-level language implementation, and even fewer have built more than one.

- *Importance of performance.* As discussed previously, the primitives will affect the performance of every program that will ever be written for the implementation, at so low a level that application programmers will be unable to do anything about it.

Taken together, these features suggest that previously developed solutions cannot be used directly, although many ideas will prove to be useful.

There are several significant areas that this work will *not* attempt to cover:

- *Control-related structures.* These are much more complicated, and bound up with language semantics. This includes displays, trails, frames, contexts, and environments.

- *Design of garbage collection algorithms.* This is another complicated area with a substantial literature; here, the algorithms will be treated as black boxes. However, the choice of algorithm and the requirements that algorithms place on representations (such as extra bits for marking) will be considered.

- *Hardware performance details.* Although caches, pipelines, virtual memory, and register windows can significantly alter the tradeoffs in data structures, the added complexity would divert attention from basic questions that should be considered first.

- *Parallelism.* Parallel hardware has some implications for data structures, but the field is too chaotic at present—there is not even any consensus on language semantics or machine architectures for parallelism, let alone implementation techniques.

Prospects for further work in all of these areas are interesting, however, and will be discussed in the last chapter. In general, this work emphasizes heap-allocated data and conventional machines.

1.6 The Rest of the Dissertation

Chapter 2 is devoted to a review of existing implementations and their primitive datatypes. During the course of this research, it has become painfully obvious that there is no comprehensive survey of runtime systems. This chapter briefly describes the data structures of about 80 language systems; mostly Lisp dialects, but including Prolog, Smalltalk, Snobol, Icon, APL, and conventional languages.

Based on this review, Chapter 3 develops a formalism with which to describe datatypes, machines, and implementations. The formalism is based on abstract data types, but specialized to be constructive rather than fully general. We also include some discussion of the pragmatic considerations that must be included with the basic definitions.

Chapter 4 sets down rules and heuristics for good designs. They are expressed in English.

Chapter 5 then goes into detail on the designer and coder programs that produce definitions usable by a specially-designed Common Lisp implementation. This chapter also includes experimental results on the consequences of different designs for some benchmark programs.

Finally, Chapter 6 summarizes the progress made in this dissertation and outlines several promising avenues for further investigation. There is also a set of recommendations for improvements to standardized Lisp dialects, suggested by the formalization of their datatypes.

CHAPTER 2

REVIEW OF DATA STRUCTURE DESIGNS

> ... we have witnessed the proliferation of baroque, ill-defined, and, therefore, unstable software systems. ... many programmers now live in a limbo of folklore, in a vague and slippery world, in which they are never quite sure what the system will do to their programs.
>
> E.W. Dijkstra, *A Discipline of Programming* (1976)

Although higher-level languages have been implemented many times, the literature includes almost no discussion or comparison of their runtime systems. This chapter is the first survey spanning a wide range of different implementations built during the past 30 years. It covers a variety of languages (though with an emphasis on Lisp), and emphasizes the internal representations of explicit data objects. (Thus the design of stack frames and other control-related objects is excluded, although they may be mentioned on occasion.) The main criterion for inclusion of a system was that information about internal data structures was available. Languages embedded in other languages, in such a way as to utilize the primitive datatypes of the base language, were also excluded. On the other hand, if part of a data structure is built into hardware, the hardware will be described. Finally, only languages with nontrivial data structure implementations have been listed; this excludes for instance most conventional languages like C and Fortran, although some aspects of PL/I, Algol-68, and Ada[1] will be of interest.

The sketchiness of the publications can be seen by the primary sources used here: appendices in manuals, working documents, personal communications, and in a unfortunate number of cases, uncommented source code. This means that there is considerable variation in the detail and accuracy of the descriptions. Since the systems described span most of the entire history of computing, there is considerable variation in the terminology. Typically the original terms will be used, along with brief definitions, both for the sake of accuracy, and to avoid confusion over connotations that the modern equivalents might not share with their predecessors. The implementations for each language are listed roughly in chronological order.

---

[1] Ada is a trademark of the Department of Defense.

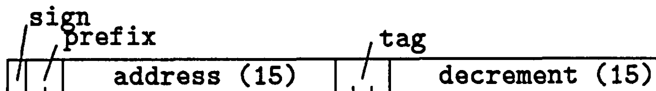

Figure 2.1. Fields in IBM 704 Word

2.1 Lisp

Lisp[2] systems are chiefly characterized by a fairly flat type space, containing 5 to 30 types all with a fairly equal status. The types are quite nonuniform in size, ranging from small objects such as characters to large high-dimensional arrays. Since few Lisp compilers do any significant amount of type inference, performance of type discrimination and dispatching is considered crucial to overall quality. Special problems in Lisp have included the extremely large numbers of small list cells, and the importance of symbols as the means by which the parts of the system are connected to each other. Also, the representation of large integers has been an issue for some dialects of Lisp.

2.1.1 LISP 1

The first Lisp system was LISP 1 by McCarthy and his students, whose evolution has been extensively described by Stoyan [153]. The data structures in LISP 1 were similar to those in the Fortran List Processing Language (FLPL) [55]. FLPL divides a 36-bit IBM 704 word into five parts: sign, prefix, address, tag, and decrement, as shown in Figure 2.1. There are functions XCSRF, XCPRF, XCARF, XCTRF, and XCDRF to access all of these fields, although LISP 1 discarded all but the CAR and CDR operations.[3] The prefix field's bits discriminated between atoms and lists, as well as between "owners" and "borrowers" of objects—the difference being whether the reference was the original one or had been acquired later on. This seems to have been used primarily for storage recovery, but the details are obscure. (This distinction was erased in all later dialects of Lisp, although it was retained for many years in other list-processing languages.)

2.1.2 7090 LISP 1.5

LISP 1.5 was the first Lisp dialect to achieve widespread dissemination. The LISP 1.5 Programmers Manual [105], a classic Lisp reference, also includes some

[2]For many years, the term "Lisp" has been treated as a name and not an acronym, although it originally derived from "LISt Processing". When discussing particular systems, the original name will be used; when referring to the language in general, I will use "Lisp".

[3]In accordance with common usage, "car" and "cdr" will be used as ordinary nouns.

of the basic facts about the IBM 7090 implementation. LISP 1.5 defines relatively few types of objects: lists, fixed- and floating-point numbers, arrays of up to three dimensions, and symbols. Although functions can be compiled, the resulting object code is not "first-class" and is permanently connected to the symbol naming the function. In turn, symbols are not completely distinct from property lists, nor are arrays clearly distinguished from functions. LISP 1.5 was eventually implemented on many machines, the first of which was the IBM 7090, a descendant of the 704 that was also a 36-bit machine with an 15-bit address space.

The basic design divides the address space into several kinds of spaces, as shown in Figure 2.2. The heap contains only list cells, all of the same size, and since the 7090 word was large enough for two addresses, a list cell requires exactly one word. (Due to details of the machine, the complements of the car and cdr are what is actually stored.) A number is actually a list cell with a negative address in the car, and a pointer to the number proper in the full word space, which is essentially the space for all 36-bit untyped data. Symbols are equated with *property lists*, which are flagged with -1 in the car, and the familiar alternating property-value form in the cdr. All parts of the symbol, including its print name and value, are on this list, in no particular order. The print name is itself a list, whose successive cars point to full word space, each word of which contains 6 BCD characters. Figure 2.3 illustrates the layout of a typical symbol. This symbol has only two properties; [20, p. 66] lists 10 properties as being commonly used by the system.

Full word space and the heap are the only areas of memory where storage can be recovered. Unallocated space is linked together into a free list (the cdr position being used for the next address, even in the case of full words). This works because the allocated objects were always one word in size. A mark-and-sweep garbage collector (GC) reclaims unused storage, using a bit table "next to full word space." The binary program space (BPS) contains both compiled code and arrays; neither are GCed. The entire process required one second.

From a modern perspective, there is much in LISP 1.5 that seems bizarre or even ridiculous. Still, the implementation is worth studying, not only because of the insights into early programming practices, but because it is one of the very few implementations that was not influenced by previously existing Lisp systems, while it influenced succeeding Lisps for many years.

2.1.3 M-460 LISP

The Univac M-460 was a military version of the Univac 490, a 30-bit machine with 32K words of memory[4]. The language was LISP 1.5, derived from the 7090 implementation by Hart and Evans, and described in [20, pp. 191-203].

Some implementation details are the same as for the 7090 system. Pointers are 15 bits, and are packed two to a word, making a one-word list cell. However, numbers are lists of one to three words preceded by a flag-word; each of these words contains only 10 significant bits, since these are the only values that are

[4]The machine was said to "have 32000 registers"!

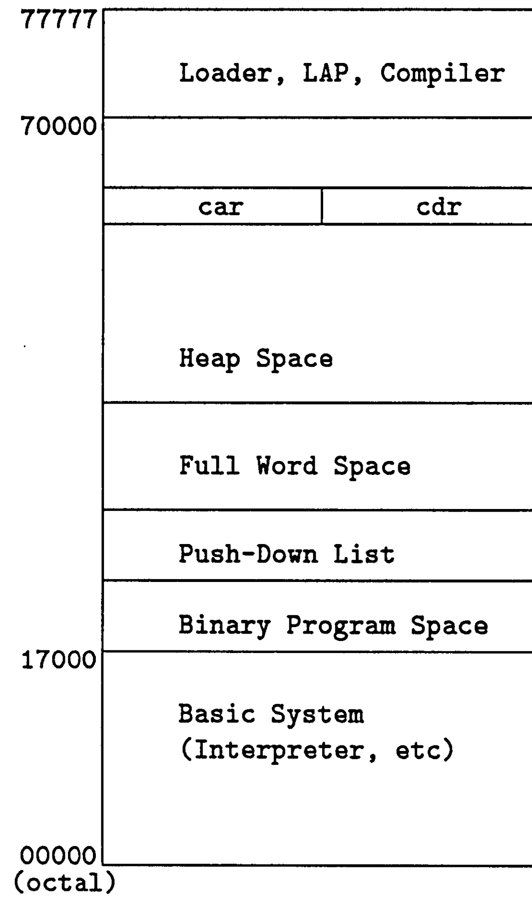

Figure 2.2. Memory Allocation in LISP 1.5

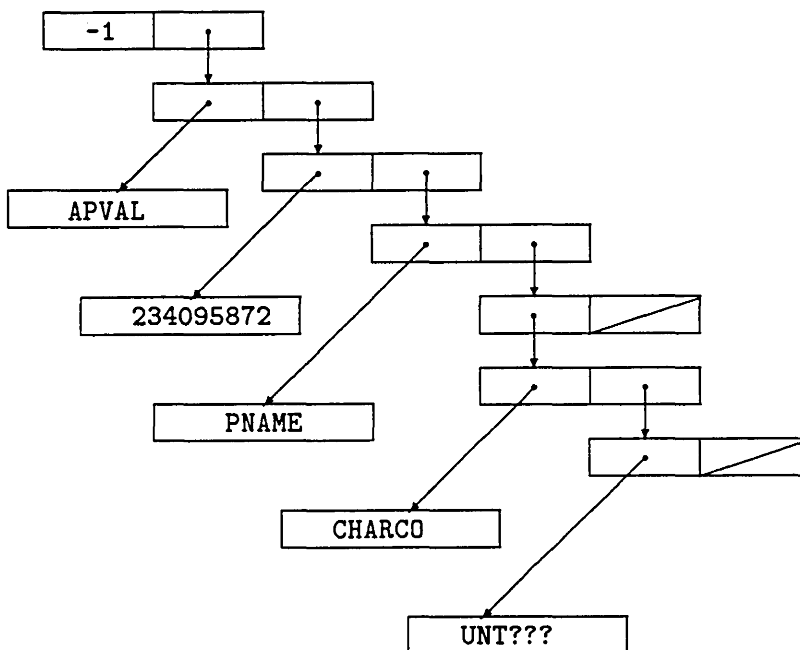

Figure 2.3. The Symbol CHARCOUNT in LISP 1.5

recognizable by the GC as numbers and not addresses (small addresses apparently point into the GC's bit table). In addition, M-460 characters are required to be 8 bits instead of the 6 allowed by the BCD character set of the 7090, so a name is represented as a list of characters (small numbers). This gave rise to perhaps the first published comparison of Lisp implementation techniques; it was observed that for a typical list of symbols, "string form" needed 530 words as opposed to 470 words for 7090 LISP 1.5, which was not considered severe, and compensated for by simplified handling and GC.

2.1.4 Q-32 LISP

The Q-32 was another early time-shared machine, built by the System Development Corporation (SDC). It had a 48-bit word and 65K words of storage, and did 1s-complement arithmetic. The Lisp was an implementation of LISP 1.5 derived from the M-460 implementation, and described by Saunders in [20, pp. 220-238]. Q-32 LISP is notable for being cross-compiled from the 7090 to the Q-32, especially since the greater word lengths necessitated special handling on the 7090 side.

Q-32 LISP's runtime structure is closer to the 7090 than to the M-460. It has a fullword space separate from free storage, a binary program space, and so forth.

Numbers have a somewhat less compact representation than on the 7090, the value being placed in a one-element array, which results in a total of 3 words being used (pointer, array header, value). Atoms have a head cell whose cdr points to the property list (the first cell of which is the print name), and whose car points to the *special* cell—essentially a binding. This is different from *SPECIAL* declarations, which are indicated by a bit in the tag. In fact, a bit in the tag field is also used to indicate if a function is being traced, with the result that the property list is not used by the system at all.

2.1.5 PDP-1 LISP

PDP-1 LISP was a subset of LISP 1.5 implemented by Peter Deutsch in 1963-64. It was notable for its small size, both with respect to the language (only 42 symbols were present initially), and with respect to the implementation (2000 18-bit words, with the ability to go up to the full 12-bit address space of the PDP-1). It did not include a compiler. The documentation is rather limited, consisting mostly of a sparsely commented assembly language listing in [20]. To make matters worse, the assembly code is written for compactness.

The data types supported are atoms (symbols), numbers (integers), and list cells. Memory consists of variable-sized list space and full word space, while types are distinguished with 2 bits in the high end of a word (the high-order 6 bits were not used for addressing).

2.1.6 LISP 1.6 and UCI Lisp

This was an important Lisp dialect in the 60s and 70s. A PDP-10 version is described in a SAIL Operating Note [124]. Most of the internal representation is identical to that for LISP 1.5, at least partly because the PDP-10 is, like the 7090, a 36-bit machine with an 18-bit address space. Memory allocation is basically the same.

LISP 1.6 internals have several differences worth nothing:

1. Definition of a special value cell that never moves during garbage collection, and is therefore directly referenceable by compiled code.

2. Strings, which are represented as uninterned symbols. (Oddly enough, the quotes around the printed representation of the string are saved along with the string proper.) Five 7-bit ASCII characters are packed into fullwords (in contrast to the 6 6-bit characters on the 7090).

3. A four-way division of numbers into *inums*, *fixnums*, *bignums*, and *reals*. Inums have an immediate representation and so are limited in range, typically to $[-2^{16}, 2^{16}-1]$. They have an offset representation that overlays the high-order part of the address space. Fixnums are the same as LISP 1.5 fixed-point numbers, while bignums are arbitrary precision integers, represented by lists of fixnums, as illustrated in Figure 2.4. (Reals (floats) are represented the same way as fixnums, but with the tag `FLONUM` instead of `FIXNUM`.)

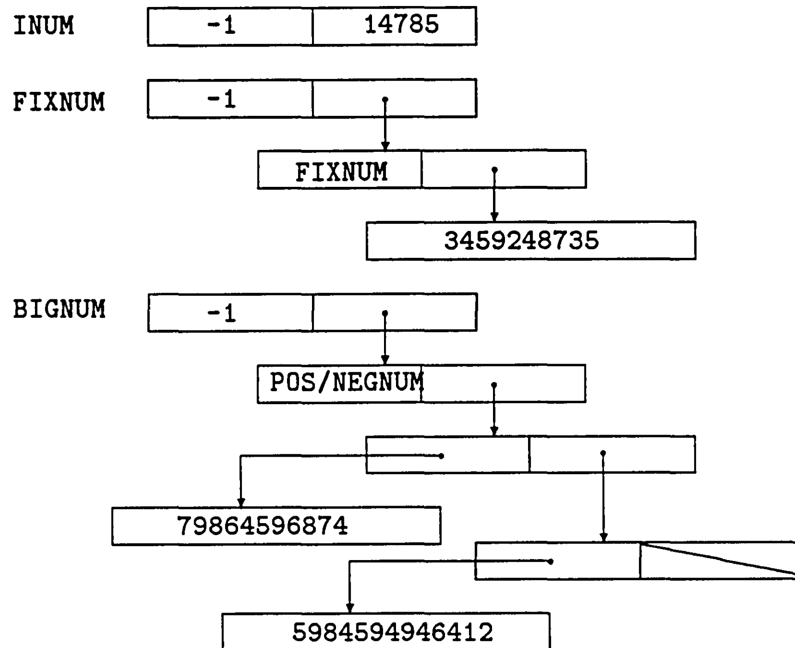

Figure 2.4. Number Representation in LISP 1.6

UCI Lisp is a compatible extension of LISP 1.6 dating from the early 70s [107]. As such, its data representations are almost completely identical to those used by LISP 1.6; most of the extensions were made at the Lisp level rather than at the machine level. Bignums appear to have been dropped, while strings are treated less like symbols (no property list and no value, although they may still be "interned" if desired).

2.1.7 LISP 2

During the mid-60s, there was some excitement over a new, redesigned successor to LISP 1.5, to be called LISP 2. The user-visible syntax was Algol-like (though S-expressions were still available), a variety of datatypes and declarations were available, and new control structures were added. Its success might have had a profound effect on the course of Lisp development, but although several papers and technical reports were written [2,95], System Development Corporation and Information International Inc. were unable to secure sufficient funding to complete an implementation for the Q-32 (the same machine as mentioned in Section 2.1.4).

Nevertheless, development did proceed to the point of data structure design, and chapter 11 of [1] is a very detailed specification. The design is somewhat complicated, mostly because of some rather unusual objects that were defined for

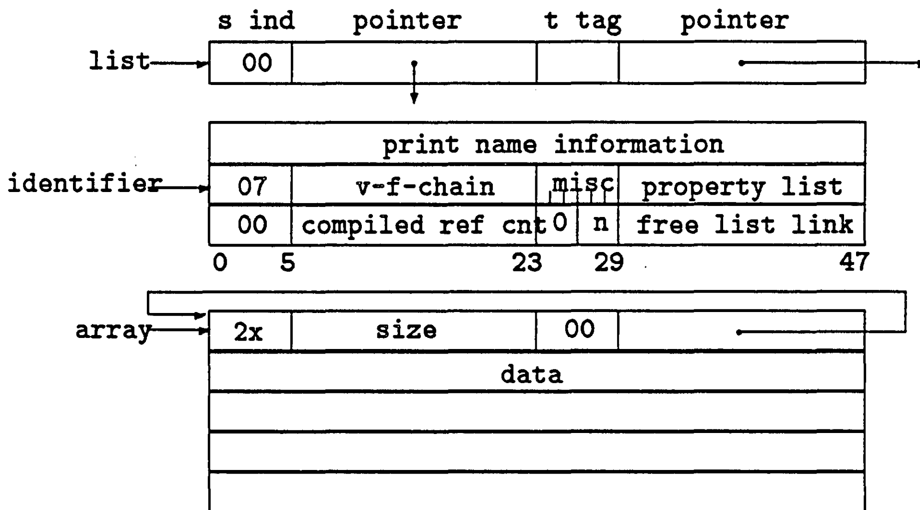

Figure 2.5. Data Structures in LISP 2

LISP 2. Over 30 types are discriminated by a 6-bit *t-tag* field stored with each object. Despite this capability, objects are also segregated into four regions: list space, array space, binary program space, and *triple* (usually means symbol) space. List space is reasonably simple, consisting of one-word list cells only (the tag for lists was 00). Array space holds numbers, strings, and *formals* (formal arguments?), as well as arrays. The first word of each of these includes a size and a self-pointer (for use in GC). Arrays actually had several tags, depending on the type of data being stored. Similarly for numbers; the same 48-bit field was considered to represent an octal, integer, or real, depending on the tag. String characters were 8-bit ASCII, and packed six in a word. (The t-tag includes a 3-bit field to tell how many characters were in the last word—the size field counted whole words only.)

Triple cells are distinct from property lists. As might be inferred from the name, triple cells consist of three words, although pointers to a triple cell normally address the middle word, which is where the type data is stored. The first word contains various data that is pointed to directly from BPS; in the case of identifiers, the first word contains print-name information. The middle word also includes the property list pointer and a pointer to the *v-f-chain*, a circular list of pointers used during evaluation. The third word includes a link to the next free triple, as well as a count of compiled code references. Figure 2.5 summarizes all this graphically. Triple cells do not move during reclamation, but are linked into a free list.

It is clear that LISP 2 was very ambitious for its time, perhaps too ambitious, considering the available hardware. Weizenbaum [163] wrote an extensive review and critique, with some remarks that are intriguing from our point of view. For instance, the variable-sized objects of LISP 2 are characterized as bad, because they would make the garbage collector more complex.

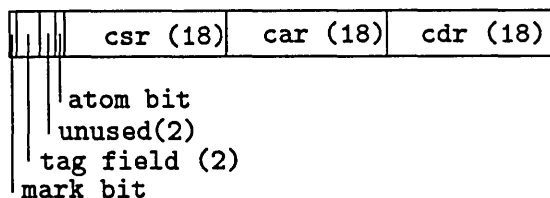

Figure 2.6. UT LISP Word Layout

2.1.8 UT LISP

UT LISP got started in 1966, as an implementation of LISP 1.5 on the Control Data 6000 machines at the University of Texas at Austin [112], and remained in use for many years [35]. The CDC 6000/7000 machines are 60-bit machines with 18-bit address spaces.

UT LISP is patterned very closely after the 7090 LISP 1.5. The address space of the CDC only requires 18-bit pointers, and since only four bits are needed for tagging, this leaves 20 bits unused by list cells, and so the decision was made to put in an extra 18-bit field, called the CSR field ("S" meaning "special"). It is completely identical to car and cdr in its behavior, and is exploited for several optimizations. For instance, each entry on the property list is a single list cell; the csr is the name of the property, the car is the value of the property, and the cdr points to the next property. This yields a 2-to-1 savings in the number of words, effectively doubling the available heap space.

The *pname* (print name) of atomic symbols uses each of the three pointers to address words containing 10 six-bit characters each, thus allowing symbols to be up to 30 characters (unused positions are filled with zeros). The symbols themselves just have pointers to their property lists in the csr fields, while the car is a self-pointer and the cdr is `nil`. Integers and floats are similar; although the csr field points to the 60-bit words with the actual data. Actually octal integers and decimal integers are tagged differently, but the only behavior difference is in printing and reading. Characters are represented by atoms with one-character names.

As in 7090 LISP, the heap is divided into list and fullword spaces, GCed using mark-and-sweep.

2.1.9 BBN LISP

BBN LISP was another PDP-1 system, and is partially described by Bobrow and Murphy [23]. The dialect basically resembles LISP 1.5.

Perhaps the most interesting feature of BBN LISP is the attempt to use a Lisp-specific virtual memory, with a drum as backing store. Of the 18-bit address space, only 16K words were main memory, while 88K were on the drum (65K of the

remainder being dedicated to immediate representation for integers). The address space between 0 and $300,000_8$ is divided into different areas for each type of data, including lists (list cells require two 18-bit words), value cells, plist cells, fullword integers, pushdown list (stack), function cells, pnames, and the reader's hash table. Each area is fixed in size, ranging from 4K to 48K words. All areas are also uniformly divided into 256-word pages, which are managed via a page table.

Each page has a separate free store list, since GC works on a page-by-page basis, so as to minimize I/O. Experimental results were that the whole scheme slowed BBN LISP by about a factor of two.

2.1.10 1108 LISP

In the late 60s, E. Norman built an implementation of Lisp for the Univac 1108, a 36-bit machine with an 18-bit address space. The Lisp dialect was basically LISP 1.5 [117].

1108 LISP only uses half of the address space, and divides it into 128-word pages, each of which is dedicated to a single type. Unused pages are linked together, as are unused objects within a page. The type of object on each page is stored in a table with one word for each page, although the code itself is only three bits. Type codes distinguish lists, integers, octals/print-name characters, floating point numbers, out-of-bound addresses, compiled code, linkage nodes (a special form of compiled code), and symbols. List cells and all types of numbers each occupy a single word. Compiled code is a block of memory with a 1-word header divided into many smaller fields, while a symbol consists of two words divided into four fields of equal size, containing pointers to value, property list, print name, and hash link (which points to another symbol in the same hash bucket—used during reading). The print name is a list of octals.

Garbage collection is basically mark-and-sweep, but the marking of numbers is unusual in that every 32nd word on a page of numbers is treated as a bit vector containing the mark bits for the next 32 numbers. (Presumably this is to allow an even division of a page, although it wastes 4 bits in each word being used as a bit vector.) Pointers are marked by complementing, and compiled code is marked by setting a bit in the header.

2.1.11 LISP F3/F4

LISP F3 and LISP F4 are implementations of LISP 1.5 written by Nordstrom at the University of Uppsala in Sweden [115]. They consist only of an interpreter, and are written in Fortran 66 [F. Botman, personal communication].

The only datatypes defined are atoms, strings, small integers, and lists. List cells are indices into car and cdr arrays, at least above a certain point. Below that the index indicates an atom and indexes a *pnameindex* array that points to the printname. The car of an atom is normally its toplevel value, but the car of a string is the symbol `LISPF4-STRING`. Small integers are indices into still another unused and unallocated part of the car and cdr arrays.

2.1.12 MicroLISP

This was the subject of a well-known paper by Deutsch, one of the earliest proposals for a specialized Lisp machine [42]. The MicroLISP language was essentially an implementation of BBN-LISP, but with a facility to define new types of structures. The ideas were later incorporated in ByteLisp and Interlisp-D.

The MicroLISP data representations are generally similar to those in BBN-LISP; objects are stored in *quanta* (pages) that each held objects of only one type, while integers in the range $[-1536, 1535]$ are permanently allocated to their own addresses. MicroLISP also has tags to distinguish integers, floats, and pointers which are used only during calculation of intermediate results (so the GC will not be confused when it scans the stack).

2.1.13 PDP-10 MacLISP

MacLISP was built in 1973. The PDP-10 implementation was highly optimized, and became famous for outperforming a version of Fortran on the same machine [45]. MacLISP includes seven basic types of objects: single-precision (36-bit) integers, single-precision floats, bignums, symbols, lists, arrays, and hunks. Hunks are essentially short vectors, and can be garbage collected while arrays are not.

MacLISP apparently originally used separate fixed-size regions for each type, but converted to allocating by pages, as described by Steele in 1977 [147]. Since the PDP-10 can fit exactly two 18-bit addresses in a 36-bit word, and since the address space is rather small, tags are undesirable, and type discrimination is based on the use of a type table indicating the *segments* (pages) devoted to each type, as illustrated in Figure 2.7. Pages are allocated to each type as they are needed. Although the technique was already known, MacLISP appears to have originated the term "BIBOP" as an acronym for Big Bag of Pages ("BBOP" is also seen sometimes). The segments are each 512 words in size, which means that the segment table need contain only 512 entries, and the index to this table is just the nine high bits of a pointer. One halfword of a table entry is bit-encoded for types and other information (read-only memory, etc), while the other half is a pointer to the symbol naming the type. The bit-encoding exploits special instructions of the PDP-10 that allow testing for several types at once; for instance, a `numberp` test is possible by enabling three bits, one for each type of number. The pointer to type name is also used in a clever way for dispatch tables, since the type names are allocated consecutively in the symbol table, and so the addresses of the symbols are all adjacent and could be used as unusual offsets.

Under this basic scheme, list cells, fixnums, and floats all occupy full words, while bignums are divided into halves, one with the bignum's sign, the other with a pointer to a list of fixnums. Symbols need both a word in the symbol segment and two words somewhere else, usually in a read-only segment. The word in the symbol segment has pointers to the symbol's property list and to the two-word block, which in turn points to its value cell, print name, count of function arguments, and some random bit fields. `Nil` is not represented as a symbol, but as 0, which is a memory location also containing 0, so `car` and `cdr` of `nil` are `nil` always, but symbol

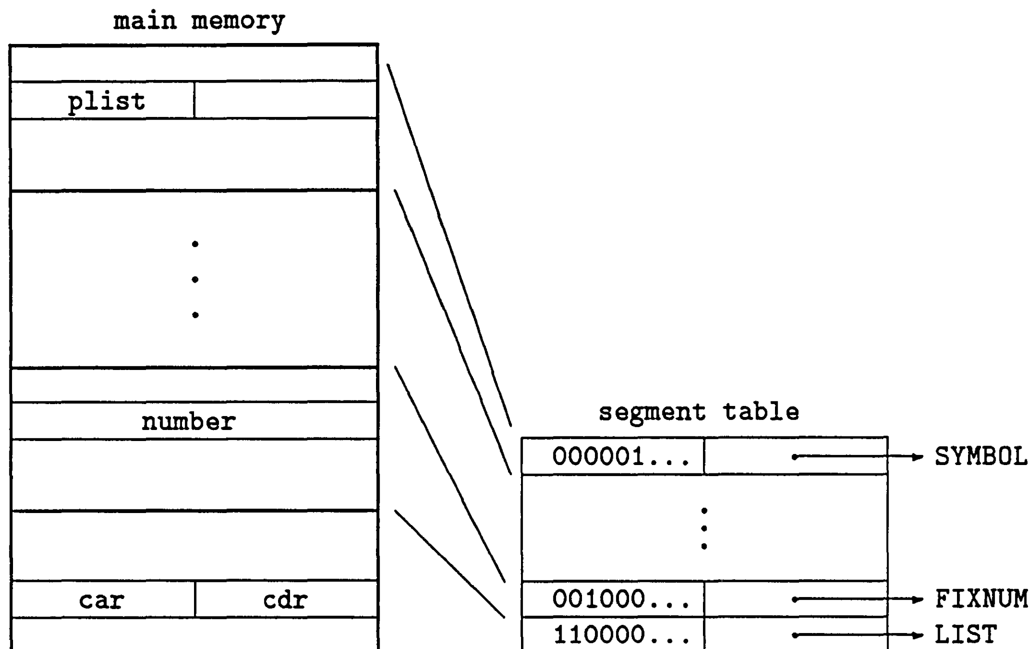

Figure 2.7. Data Representation in PDP-10 MacLISP

functions always need to do `nil` tests. Hunks are always allocated in power-of-2 sizes.

Since MacLISP cannot change the type of a segment once it has been assigned, and since the available memory is limited, if space has been exhausted, it will attempt to GC *before* requesting a new segment from the operating system. GC is mark-and-sweep, with the mark bits living in *bit blocks* that have their own segments.

2.1.14 Multics MacLISP

MacLISP on the Multics system was similar in many ways to its incarnation on the PDP-10, although it was built by a different group [B. Greenberg, D. Moon, personal communications]. The hardware of the system was the GE 645, a 36-bit machine with many 72-bit operations, a nominal 72-bit address space, and a variety of registers with lengths from 18 to 72 bits. A curious feature of Multics MacLISP was its use of PL/I in a number of places internally.

A pointer is a 72-bit object, with a variety of fields, as illustrated in Figure 2.8. The 9 types are bit-encoded as in PDP-10 MacLISP, but cons cells had a type code of 0 (no bits turned on), in order to speed up pointer-chasing. Fixnums use the second word for a 36-bit value, as do flonums. Atomic symbols consist of a 2-word value pointer, 2-word plist (property list) pointer, and a PL/I-like varying-length string of characters stored contiguously with the symbol.

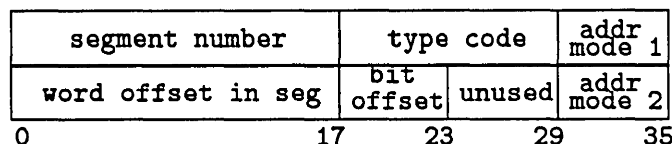

Figure 2.8. A Multics Maclisp Pointer

Cons cells are stored in a separate heap, cars first, then cdrs. All other objects go into array space. GC is stop-and-copy.

2.1.15 Interlisp-10

There are several implementations of Interlisp [156], all of which share the same "Interlisp Virtual Machine" described by Moore [111], which itself specifies very little about the representation of basic types. It does require 11 basic types and a facility for adding new types. The user-defined types all look like structures with fields. Interlisp-10 is the original system, which evolved from several earlier efforts, most notably BBN-LISP.

Section 3 of [156] supplies a basic description. List cells are handled in the same way as for PDP-10 MacLISP, namely the car and cdr both fit into a single word. Literal atoms (symbols) are three words in length. The first word includes the property list and top level binding (which can be accessed via car and cdr functions). The second word is an instruction that calls function code if defined, and the third word includes pointers to the *pname* (print name), and a reserved half for an extension to reference the file containing a function's definition. The pname is a raw string object—a block of words with 7-bit characters packed 5 to a word. The first character contains the length in characters; since it includes itself in the length, the maximum length of a pname is 126 characters.

Large integers are allocated in one word of storage, and thus fall in the range $[-2^{35}, 2^{35}-1]$, overflow past this range resulting in failure. (Interlisp systems did not include bignums until recently.) Small integers fall in the range $[-1536, 1535]$, and their representation is immediate, but offset by a constant. Floats are allocated in one word, in the standard PDP-10 format.

Arrays are somewhat complicated, since they have different subregions storing different types of objects. The array header includes a length, a half-word for GC purposes, and offsets to the pointer and relocation subregions of the array. The first section following the header contains *unboxed* data (36-bit untyped words).

Following this is the pointer subregion, then the relocation information area, which specifies which of the unboxed entries should be modified if the array is relocated.

Strings are divided into *string pointers* and *string characters*. String pointers allow the sharing of characters in memory, such as for substrings (Interlisp has no destructive operations on strings). A string character is one word containing 5 7-bit characters. Sequences of string characters may appear in memory together. String pointers are divided into a 15-bit length and a 21-bit pointer to string character and character within it.

Interlisp-10 allocates space by pages, with each page storing distinct types of objects. A type table stores the type associated with each page. Fixed-length types such as numbers are no special problem, but variable-length objects require groups of contiguous pages. As with MacLISP, GC runs before the allocation of new pages. Freed fixed-length objects are collected into a free list, while variable-length objects are compacted. GC is always done for fixed-length types, while a variable-length type is only collected when it alone is exhausted. There are also some dynamic heuristics governing how many extra pages to allocate to types.

2.1.16 LISP-11

LISP-11 was written by Jeffrey Kodasky for the PDP-11 during the mid 70s [89]. The dialect is LISP 1.5 with a handful of extensions, mostly primitives to address memory directly. The implementation was written in assembly language to run under RT-11, a primitive multi-tasking system.

LISP-11 divides memory into free space (heap), array and I/O buffer space, and various buffers for code, stack, and so forth. It then allocates all of the free space in units called *cells*, each of which is two 16-bit words in size. The first word is the car, the second is the cdr or a raw word. All pointers are cell-aligned, so this mean two bits are unused at the low ends of both car and cdr. The spare two bits of the car are a mark bit and a pointer/word bit controlling the interpretation of the cdr field. If the pointer/word bit is 1, then the cdr is a raw 16-bit word, otherwise the cdr contains a 14-bit pointer and the 2-bit field distinguishes literal atoms, literal strings, and lists. (See Figure 2.9.) The car of an atom points to a linked list of two-character cells linked together via their car fields, while the atom's cdr points to the property list—basically the same as for LISP 1.5 on the 7090.

2.1.17 ULISP

ULISP is another PDP-11 Lisp, modelled after the Univac 1100 Lisp by Norman (see section 2.1.10). It was written by Robert Kirby at the University of Maryland [85]. Along with some forgotten operating systems, ULISP could be run under Sixth Edition Unix, thus making it the first Unix Lisp system. The dialect bears some resemblance to LISP 1.5, but includes more primitive datatypes, including both single and double precision floats.

The basic pointer is a 16-bit value. Like the Univac Lisp, the data area of ULISP is divided into pages, of 1024 bytes each, each dedicated to a single type.

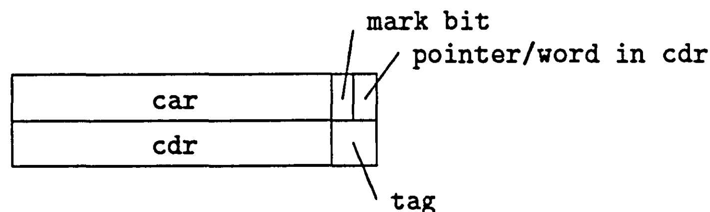

Figure 2.9. Data Representation in LISP-11

The page table contains one-byte codes for each type; the codes themselves are all even numbers, to allow use of the type code for dispatching tricks. Car is stored first in memory, then cdr, but the pointer to a cons cell actually points at the cdr and uses a predecrement addressing mode to address the car.

2.1.18 Cambridge LISP

Cambridge LISP is a British development originally undertaken by Fitch and Norman [46]. It is still in use, and now includes commercial microcomputer versions. The dialect is largely based on Standard LISP [102], with additional concern on the designers' part relating to error handling for limited storage space, and rational numbers. The original implementation was written in BCPL for the IBM 360/370 machines, and has since been ported to the GEC System 63, Acorn 32016, Acorn ARM, and the Atari ST. Despite the variety of ports, the internal structure has remained essentially the same [J.P. Fitch, personal communication].

The implementation is tagged, with 32-bit pointers divided into a 24-bit address and an 8-bit tag field (which matches exactly with both the 360/370 and 68000 processors). Cambridge LISP is notable for its careful assignment of tags to types, illustrated in Figure 2.10. The ordering essentially amounts to a topological sort [88] of the type hierarchy, which means that many type tests are comparisons. For instance, `numberp` is true if the tag is positive but less than 4. One disadvantage of this assignment is that FF (-1) cannot be used to tag negative small integers, but since FF does not tag any other type, untagged negative integers may be examined during GC without any problem.

The heap and stack grow toward each other in the same space, which justifies the use of a compacting collector. `Nil` is at the very beginning of the heap.

2.1.19 CLisp

CLisp was built for VAX/VMS systems at the University of Massachusetts in 1977-78. Types available included arrays, bignums, compiled functions, files, fixnums, flonums, vectors, arrays, symbols, and cons cells. It was distributed to a

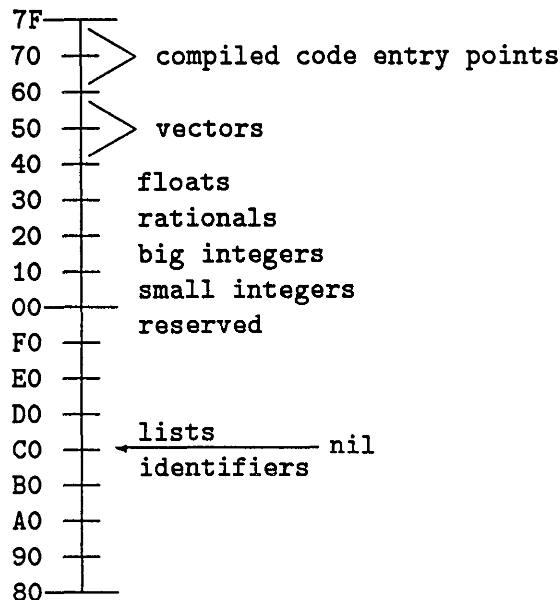

Figure 2.10. Tag Assignment in Cambridge LISP

number of sites [D. Corkill, personal communication]. The kernel (runtime system) was hand-coded assembly language.

The basic object is 32 bits; either an immediate fixnum (flagged with 1 in the least-significant bit), or a longword-aligned pointer. Pointer types are discriminated by 512-byte pages (BBOP), which is the page size built into the VAX hardware. Large objects could extend over multiple pages. Flonums (floats) are always 64-bit numbers. Cons cells stored with car first, then cdr. Strings start with a 32-bit length (in characters), are zero-terminated, and always allocated in multiples of four bytes. Vectors are similar, but with both length in bytes and in number of elements. Symbols have four components: value/function, print name (a string), hash table link (another symbol), and pointer to the property list. NIL is a symbol, with an address of 0. Arrays consist of a 32-bit length field, followed by a descriptor in the standard VMS format and the data itself. A variety of specialized arrays are available. GC is mark-and-sweep, with a stack-allocated mark bitmap.

2.1.20 ByteLisp

ByteLisp was an implementation of Interlisp for the Alto. A group led by Peter Deutsch at Xerox PARC worked on it [41]. The Alto was a 16-bit machine with from 64K to 256K of memory available.

Despite the limited real address space, ByteLisp defines a virtual address space of 2^{24} 16-bit words, although only 2^{22} are actually used. Many data types are

allocated from areas of fixed size and position, but others (list cells, large integers, string and array descriptors, floats, and user-defined datatypes) all share a single heap. The fixed partitions are sometimes exploited by using shorter (usually 16-bit) pointers. The heap is organized into 512-word pages, each containing objects only of a single type.

List cells have a compacted format (*cdr-coding*), where each cell is 32 bits, broken into a 1-bit F field, a 7-bit Q field, and a 24-bit P field. The P field is generally a full pointer to the car, while the Q field usually refers to a nearby cdr; the F field decides how the P and Q fields are actually interpreted. The compaction is very good, with the average size of list cells being 34-35 bits (worse case would be 64-bit list cells). The three types of integers include those in the range $[0, 2^{16} - 1]$, the range $[-3 \cdot 2^8, -1]$, and 32-bit heap-allocated integers. Symbols are stored as separate tables for function, "permanent" value, and property list, while the print name is compacted in a complex fashion. Reclamation is based on reference counting, as described in [43].

2.1.21 Interlisp-VAX

Interlisp-VAX, as described in [17], implements the Interlisp Virtual Machine in about 12,000 lines of C and assembly language. The operating system is Berkeley Unix.

It uses a BBOP scheme with rather large *sectors* (pages) of 64K bytes apiece. A sector table contains 16-bit *data type numbers* that index another table describing each type in more detail. A data object is then represented either as a pointer directly into a sector, or to a *sequence descriptor*, in the case of variable-length objects like strings. Some user-defined objects may contain a combination of pointer and immediate data; to handle these cases, the type descriptor includes both a length of the whole object and the number of pointers in it. In order to support larger immediate integers than allowed by a 64K sector, the high-order half of the address space (2^{31} to $2^{32} - 1$) is used as a representation of 31-bit integers. This is effectively a 1-bit tag with a value of 1.

2.1.22 Interlisp-D

Interlisp-D is externally similar to the other Interlisps, but internally, it was redesigned for the Xerox 1100 series of microcodable workstations.

As described in [53, pp. 73–75], a pointer is 24 bits in length. The address space is composed of 512-byte *quanta* (pages). Small integers (in the range $[2^{16}, 2^{16} - 1]$) have an immediate representation, while integers up to 32 bits are *boxed* (are allocated heap space and referred to by pointer), as are floats, which are in 32-bit IEEE format. Cdr-coding (see section 2.1.20) is thoroughly built in; cons cells are 32-bit objects, normally 24 bits for the car and 8 bits for the cdr, which suffices to address cdrs in the same page. The correct escape code converts the cons cell into a forwarding pointer to a 64-bit cell with full car and cdr pointers, but this case is supposed to be rare. Strings and arrays are allocated from a separate area.

Interlisp-D uses reference counts for reclamation, which are kept in a hash table elsewhere, and which is claimed to be sparse.

2.1.23 Zetalisp/Symbolics 3600

Zetalisp originated as Lisp Machine Lisp, which was developed from MacLISP, but is a much larger language. Many parts of the Common Lisp design were first tried in Zetalisp. Moon [109] has written an extensive overview of data structures in Zetalisp.

The Symbolics 3600 design (originally derived from the MIT Lisp Machine [18]) supports tags in hardware, which means that many primitive operations exhibit some concurrency and can be quite fast. It is basically a 36-bit machine with a 28-bit address space (of 36-bit words, not bytes). A word can be broken down in several different ways. An *object reference* has a 2-bit *cdr code* which implements cdr-coding (see section 2.1.20), a 2-bit *major tag*, and possibly a 4-bit *minor tag.* Small integers and IEEE single-precision floats use only the major tag, thus they are each 32 bits, while pointers also have a minor tag, leaving 28 bits. Figure 2.11 illustrates some of these combinations.

More complex objects such as arrays also have a header word that can have several different formats. For instance, the array header word consists of another 6-bit tag and 28 bits of type and length information, followed by the array data. Specialized arrays such as strings are packed. Function objects are quite complex. The header word has a tag and a size, followed by an additional three words of various info. Then there is a table of constants and external references (essentially a local symbol table), followed by the instructions, which are tagged as a distinct type of data.

The GC method has been publicly described [110]. It is based on a notion of ephemeral and static objects, and attempts to minimize VM thrashing.

2.1.24 Scheme Chips

In 1978-79, Steele and Sussman developed a pair of Scheme processor chips [149]. The first was a toy design; it was only a 11-bit machine, of which 3 bits constitute a tag distinguishing both data (lists and atoms) and program structures (function application, conditionals) from each other. The second effort, dubbed the Scheme-79 chip, was a more plausible design; a 32-bit machine, with a 7-bit tag field, a 1-bit mark field for GC, and a 24-bit data field. It worked basically as an interpreter on program structure objects. Although this chip was tested and found equivalent to a KA-10 running compiled Lisp, the project was not continued further.

2.1.25 NIL

NIL is a successor of MacLISP and Lisp Machine Lisp, and was one of the main influences on Common Lisp. It was only implemented for the VAX. The following description is from [53].

NIL is tagged, with a 32-bit pointer divided into a 3-bit high-order tag field, a 27-bit data field, and a 2-bit low-order tag field (see Figure 2.12). Since integers

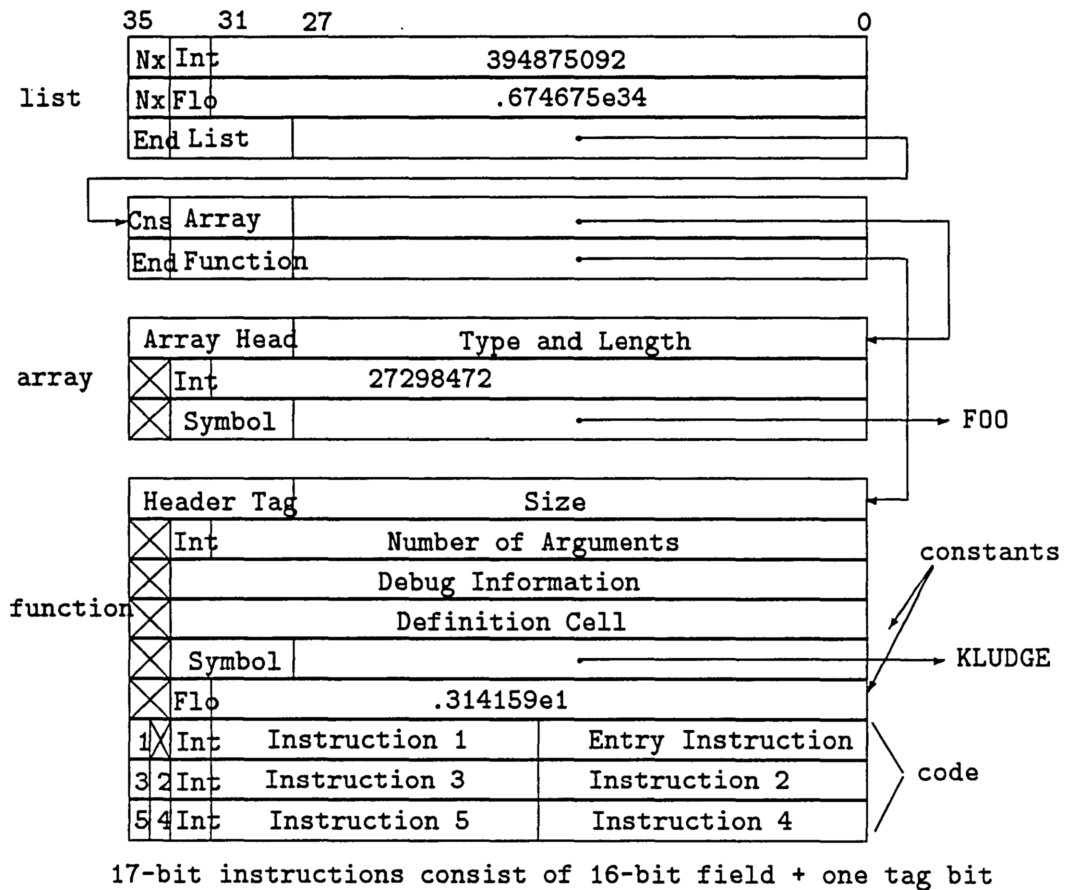

Figure 2.11. Data Representation in Zetalisp

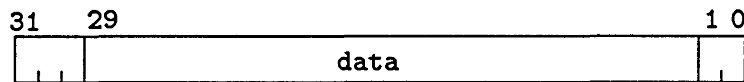

Figure 2.12. Data Representation in NIL

are represented with a zero in the low-order tag bits, and a don't-care in the high-order bits, NIL has 30-bit fixnums. Among other desirable qualities, the integer can be used directly as an index to a general vector. NIL uses different tags for stack-allocated and heap-allocated objects, and sets up the tags in such a way that stack-allocated objects have the correct address for the dedicated stack space of the VAX. That is, the high-order bit of a stack-allocated version of a type is 1, which is the stack area defined by VAX hardware. Another way to look at this is to recognize the most-significant bit as a stack/heap flag, and that there are really only four tag bits for discriminating types.

2.1.26 FLISP

In the late 70s and early 80s, the Utah Symbolic Computation Group (led by Hearn and Griss) experimented with a variety of Lisp systems, with the goal of making a portable Lisp platform for the REDUCE algebra system. One of the subgoals making this more difficult was the intent to include the Portable Lisp Compiler (PLC) [64]. The dialect to be implemented was always Standard LISP [102], an extremely small Lisp, but the first standardized dialect.

FLISP is a Fortran-based system written in SYSLISP [62]. SYSLISP was originally billed as a BCPL-like or C-like Lisp dialect; in effect it is an "unsafe" Lisp in which everything is implicitly a machine word, and any operation can be done on any object. The PLC is capable of compiling SYSLISP into reasonably efficient code; in the case of FLISP, it was adapted to compile a SYSLISP-coded interpreter into Fortran.

The data representation is somewhat abstracted. Each *item* has a `TYPE` and an `INFO` part. The low-level system description defines how these are to be represented; for instance, it is mentioned that the DEC-20 implementation of FLISP uses 9-bit `TYPE`s and 18-bit `INFO`s, leaving 9 bits for the GC to use. (Page 9 of [62] suggests that this setup has the possibility for different data representations, but this capability was apparently never exploited. Successor systems such as Portable Standard Lisp always used tags.)

2.1.27 Franz Lisp

Franz Lisp is a dialect developed at UC Berkeley to run under Unix [48]. It includes 14 primitive datatypes, including several kinds of array-like structures. The language itself is fairly small, although many optional libraries are available. Franz Lisp has been implemented on the VAX and on many 68000 systems. The description of internals here partly derives from the manual, and from the source code for the runtime system, which is almost entirely in C.

Franz Lisp uses a BBOP representation with 512-byte pages. Numbers in the range $[-1024, 1023]$ have immediate representations, while all others are boxed. (Franz Lisp is unusual in supplying and documenting functions to destructively modify the boxed representation.) Bignums are lists rather than vectors of fixnums, but with a different tag to avoid any type-aliasing. The cdr of a cons cell is stored at the lower address. Structures such as symbols and arrays are defined using fairly ordinary C structs.

The GC is mark-and-sweep. Unused string space may or may not be recovered, depending on the setting of an option while building the system. The rationale for this is that string space collection is relatively slow, and that many applications do not discard enough strings (particularly when they appear as names of interned symbols, which are never destroyed) to make string recovery worthwhile.

2.1.28 Portable Standard Lisp

Portable Standard Lisp (PSL) [63] is essentially a runtime system based on the Portable Lisp Compiler (see section 2.1.26). It is written in SYSLISP with a small amount of assembly language and perhaps an interface to the operating system written in an appropriate high-level language (C for Unix, Pascal for Apollo, Fortran for Cray, etc). Most of the description is to be found in the Implementors Guide [65] and in unpublished notes.

Despite the variety of machines to which PSL has been ported, its basic structure is the same everywhere; all objects are tagged with at least 5 bits distinguishing 19 primitive types. Tags 0 and -1 must be used for positive and negative small integers, respectively; otherwise there are no special assignments. User-defined types are all built on the type *e-vector*, which is like a normal vector, but with a different tag and no functions to access from interpreted code. All objects reside in a heap, with the exception of compiled code and a few constant objects, which are in a non-GCed area known as Binary Program Space.

Table 2.1 shows how the minimum tag requirement has been met by various implementations of PSL. All use high-order tags; despite the apparent portability of PSL, many parts of system code would fail if the tags were not at the high end of the word. Of these, only the Cray-1 has unused bits, which is understandable, since the address space is only 24 bits while the normal word size is 64 bits (the extra bits are used as a relocation address for a compacting GC).

Machine	Word	Tag	Data
DEC-20	36	5	31
VAX	32	5	27
68000 (Apollo DN300)	32	8	24
68020	32	5	27
IBM 370	32	8	24
Gould SEL	32	5	27
Cray-1	64	5	37

Table 2.1. Tag and Data Sizes of Various PSL Implementations

2.1.29 FLATS

FLATS is a Lisp machine developed by a large group in Japan [60]. The supported dialect appears to be a small one resembling Standard LISP. Special types of objects include *big floats* (arbitrary precision floating point), two kinds of fast lookup tables (like hash tables), and *H-type* data, which is based on the idea of hashing CONS.

The basic data format is a 32-bit word divided into an 8-bit tag and 24-bit field for addresses and short integers. 2 bits of the tag are used for cdr coding, 1 bit flags short floats, while 5 bits are the main tag field, which distinguishes about a dozen types.

2.1.30 LeLisp

LeLisp is another highly portable Lisp system, developed at INRIA by Chailloux, Devin, and Hullot [32]. It has been implemented on at least a dozen different machines. The implementation is based on a low-level virtual machine called LLM3, which is very close to machine language. The interpreter is quite fast, as is the compiler.

Data representation is done by dividing memory into *zones* (spaces), one each for symbols, conses, strings, vectors, and floats. The contents of strings and vectors go into a *heap zone*. Short fixnums (16-bit) are not boxed (allocated), while arbitrary precision rationals are represented as trees of fixnums in the cons zone. Symbol structures include slots for function, value, function type, print name, and property list. All zones are GCed, the heap in particular is also compacted, using mark-and-sweep.

2.1.31 Tandem Lisp

Tandem Lisp was an implementation written by John Cowan and Paul Pedersen for the Tandem NonStop II fault-tolerant minicomputer, although it did not actually use any of the fault-tolerant capabilities. [J. Cowan, personal communication]. The Tandem architecture is similar to the PDP-11 (16 bits, separate instruction and

data spaces), but with data space of 128K bytes. Addressing is slightly peculiar; pointers can either be *byte pointers* addressing only the first 64K of data space, or *word pointers* capable of addressing the entire 128K in 2-byte increments.

The Lisp dialect is very simple: conses, symbols, and 16-bit signed fixnums. A cons cell is two 16-bit words, with the cdr stored first. A fixnum also requires two words, with the cdr being a magic value and the car the value of the fixnum. Symbols are structures in the implementation language (which was Algol-like). In addition to the usual property list and value/function cell, there are two bits to flag interned symbols and to indicate the presence of unusual characters in the print name. The print name is packed into bytes immediately following the symbol, preceded by a word giving the length of the name.

Symbols are allocated going upwards in data area, and conses/fixnums allocated downward (which is OK, because they are always addressed as words). The uniformity of objects, and the non-GC of symbols, makes for a rather simple mark-and-sweep garbage collector, which uses the low bit of the cdr as a mark bit.

2.1.32 T

T is a superset of Scheme with many Common Lisp features, originally described by Rees and Adams [127]. T has been ported to the VAX and to 68000-based machines.

Each object has a 3-bit low-order tag, and all objects in memory are aligned on 8-byte boundaries, which is somewhat coarse, but avoids any need to shift pointers in the data field. Only fixnums, pairs, floats, and strings merit unique tags; () and characters share a tag, while user-defined data structures are all distinguished with a "type template". Fixnums get the zero tag. Tag stripping of cons cells is eliminated by clever indexing, in which the value of the tag offsets the addresses of the car and cdr, as illustrated in Figure 2.13.

2.1.33 Spice Lisp

Spice Lisp was originally intended as a MacLISP successor to run on personal workstations. Ultimately it became a chief contributor to, and a model implementation of, Common Lisp [148]. The internal structure of Spice Lisp is almost completely described in a single document [167]. Although Spice Lisp was originally designed to run on the Perq workstation, the Perq was a fairly conventional 32-bit microcodable machine with virtual memory.

Spice's data representation utilizes a "space-tag equivalence", wherein the entire 32-bit address space is divided into 32 contiguous blocks, each of which is devoted to a single type. This means that the most-significant 5 bits of a valid pointer is also a standard type tag. Thus, object pointers may be dereferenced without removing the tag, but examination of the tag requires only a single masking operation, thereby achieving the advantages of both separate space and tagged representations. The disadvantage is that the entire address space must be available to the Lisp system,

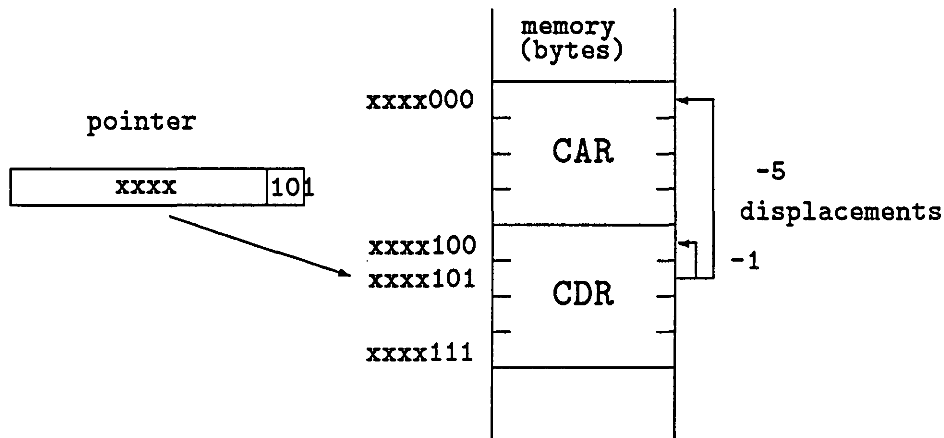

Figure 2.13. Use of Low Tags in T3

and the virtual memory system must deal well with highly fragmented programs. Areas of memory devoted to immediate types like fixnums and characters will always remain empty. Each space is also limited to 2^{27} bytes (2^{25} or about 32 million objects), but this is unlikely to be a problem. Figure 2.14 illustrates the representations in Spice Lisp.

More recently, Spice Lisp has been renamed CMU Common Lisp and been implemented for the IBM RT PC [106]. Data representations are largely the same as for the Perq, although the tag assignments have been altered—while Spice originally assigned fixnums arbitrary tag values, in CMU Common Lisp they get the tags 0 and -1 (or 31). Also, the subtypes of numbers and arrays have been gathered into contiguous ranges, which could be exploited by doing range tests on tags rather than individual tests for each subtype.

2.1.34 Data General Common Lisp

This is a derivative of Spice Lisp, briefly described by Gabriel [53]. It runs on Data General's MV family of computers, which are basically 32-bit machines, but the highest four bits of the address are treated specially by both the hardware and the operating system (three for a ring protection scheme, and one for indirection).

The objects are 32 bits long, and discriminated using a BBOP scheme with 32Kbyte segments (pages). Thus pointers are divided into two equal halves (the highest bit being ignored however), the upper half indexing the type table. Fixnums are 28 bits, where the indirect and ring protection bits tag the fixnum.

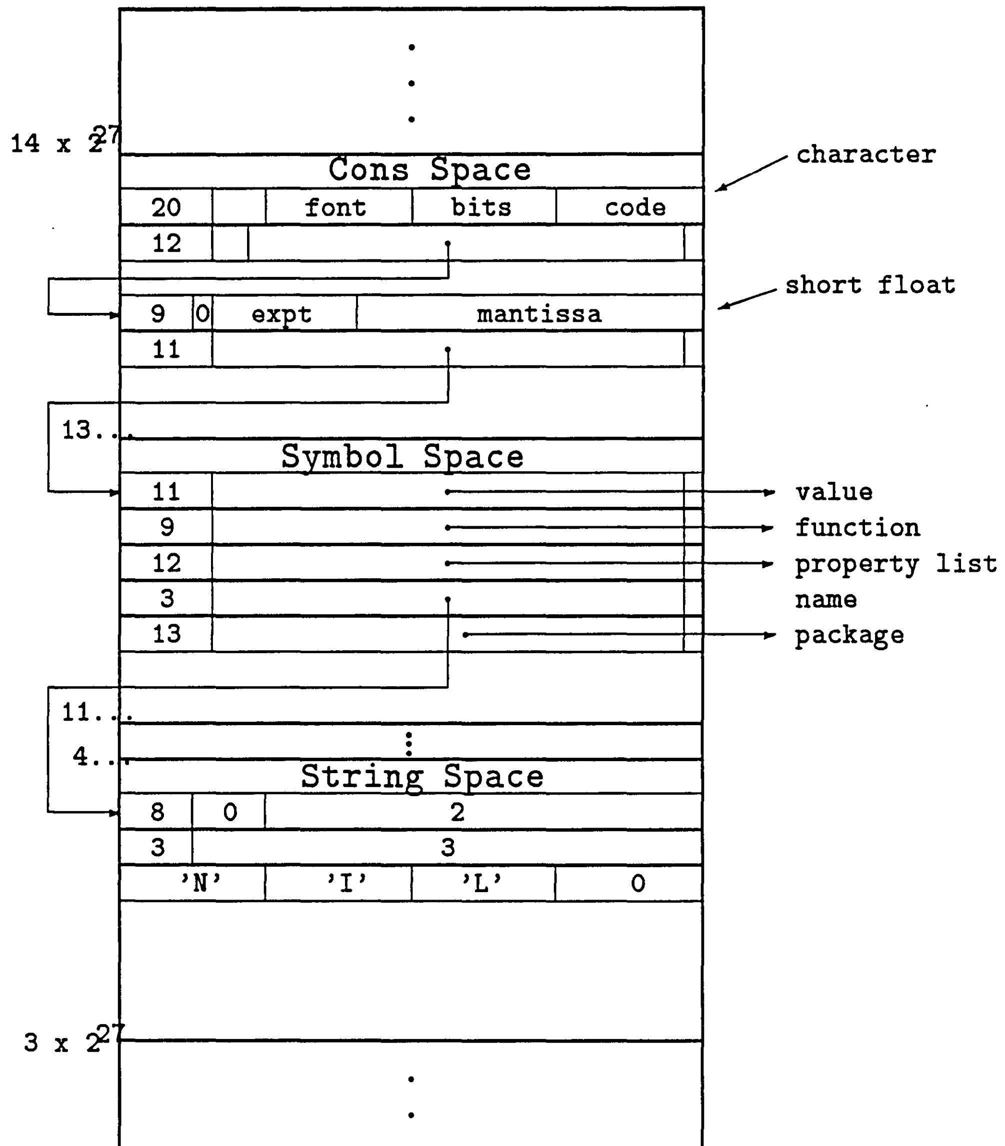

Figure 2.14. Data Representation in Spice Lisp

2.1.35 S-1 Lisp

The S-1 Mark IIA is a supercomputer developed by Lawrence Livermore Labs. It is a 36-bit machine with a 31-bit address space, and some support for tags. Nine of the tags are reserved by the hardware. A variety of numeric formats are available, up to 144-bit floats and 288-bit complex numbers, and many rounding modes are in hardware. The S-1 also has some extremely complicated addressing modes. Several papers were written about S-1 Lisp [24,25], and Gabriel describes it [53].

S-1 Lisp is based (unsurprisingly) on 5-bit tags that do almost all type encoding, with the exception of specialized array types (available for nearly every type of number), whose element types are stored with the array. Fixnums get the tags 0 and -1.

Memory is divided into dynamic, static, and read-only areas. Read-only objects can be created only, while static objects can also be modified (but not reclaimed), and dynamic objects can also be reclaimed. Compiled code goes into a subarea different from that for other data. Each of these subareas is divided into variable-size *segments* which themselves are composed of 64K byte *segmentitos*. A table in static space identifies the kind (dynamic/static/read-only) and current status of each segmentito. The GC works only on dynamic storage, and basically copies data between old and new subspaces of dynamic storage.

2.1.36 Kyoto Common Lisp

Among Common Lisp implementations, Kyoto Common Lisp (KCL) is distinguished by its use of C as the compiler target code (as well as for most of the runtime system), and by its development independently of other Common Lisp efforts [171].

The basic objects are defined as C structs, and are not particularly compact. The type field only needs to distinguish 27 primitive types, but occupies an entire C short integer (which is typically 16 bits). The mark bits for GC are in another short integer, and the components of the object are usually stored as long (32-bit) integers. This means that even fixnums and characters require storage space, but this is not as bad as it might seem; the extensive C coding in the runtime system means allocation is unlikely within a primitive (this is hard to avoid when doing Lisp-in-Lisp). `Nil` is a normal symbol. Every type in KCL has a C struct; no bootstrapping is done using `defstruct` or an object facility, as is typical in most Common Lisp implementations, especially for the "higher-level" types like hash tables and random states.

GC is mark-and-sweep, using a 16-bit (!) mark field. Each type of object has its own areas of memory, which simplifies compaction and allows reporting on space usage and recovery.

2.1.37 HP Common LISP

Hewlett-Packard's Common Lisp [70] for the HP-9000 series workstations was derived from a combination of PSL and Spice Lisp code [137]. The workstations are based on the different members of the 68000 family, and the implementation was

affected by progress through the several generations of chips, with ever-increasing address spaces.

The basic design uses 4-bit high-order tags that are effectively automatically removed by the combination of virtual memory hardware and operating system support. Representations of vectors and other objects are generally similar to those in PSL, in fact the knowledgeable user can find a package nicknamed `psl` with many of the most primitive PSL functions therein. The GC is stop-and-copy.

2.1.38 Extended Common Lisp

Franz Inc. was founded by several implementors of Franz Lisp. In addition to supporting and enhancing Franz Lisp, they produced an implementation of Common Lisp, called Extended Common Lisp or ExCL. ExCL has been ported to a variety of machines. [J.K. Foderaro, personal communication].

The data structures are tagged with three bits in the low end of a 32-bit word. Types getting their own tags are fixnums, symbols, characters, conses, and `nil`. All other objects are pointers to blocks whose first byte indicates the type. Symbols have a sixth word in addition to the five mandated by Common Lisp; it includes various flags, and a 16-bit hash value, to avoid expensive recomputation.

`Nil` has a clever representation. Since in Common Lisp, `nil` is both a symbol and a list, all of the symbol operations and all of the list operations must work correctly on it. For instance,

```
(symbol-name nil) => "NIL"
(car nil) => nil
```

This is a problem, because the basic operations should be opencoded for best performance, but the desired machine code is a displaced memory access and nothing else. This precludes the use of any type tests, so `nil` must be organized to look like a normal symbol *and* like a normal cons cell. Needless to say, tight constraints are imposed on symbol and cons representations, as well as on `nil` itself.

Figure 2.15 shows how the problem is solved. The value slot of a symbol and the car slot of a cons are at the same offset. Since both are defined to be `nil` again, initialization need only install a circular pointer. Another one can be installed in the cdr slot, which is at the same offset as the package slot of a symbol. The cdr of `nil` is also `nil`, which is not a valid package. Fortunately, accessing the package of a symbol is not a frequent operation, and can include the type test.

2.1.39 Lucid Lisp

Lucid Lisp is an implementation of Common Lisp sold by Lucid, Inc.; it is highly portable, and available on a wide variety of machines (Sun, Prime, HP, etc). Because of this, Lucid Lisp varies slightly from machine to machine, although the basic scheme is held constant [E. Benson, personal communication].

The basic object reference is a *word*, typically about 32 bits in size. The 3 least-significant bits are the *primary data type tag*, and the next 5 bits may

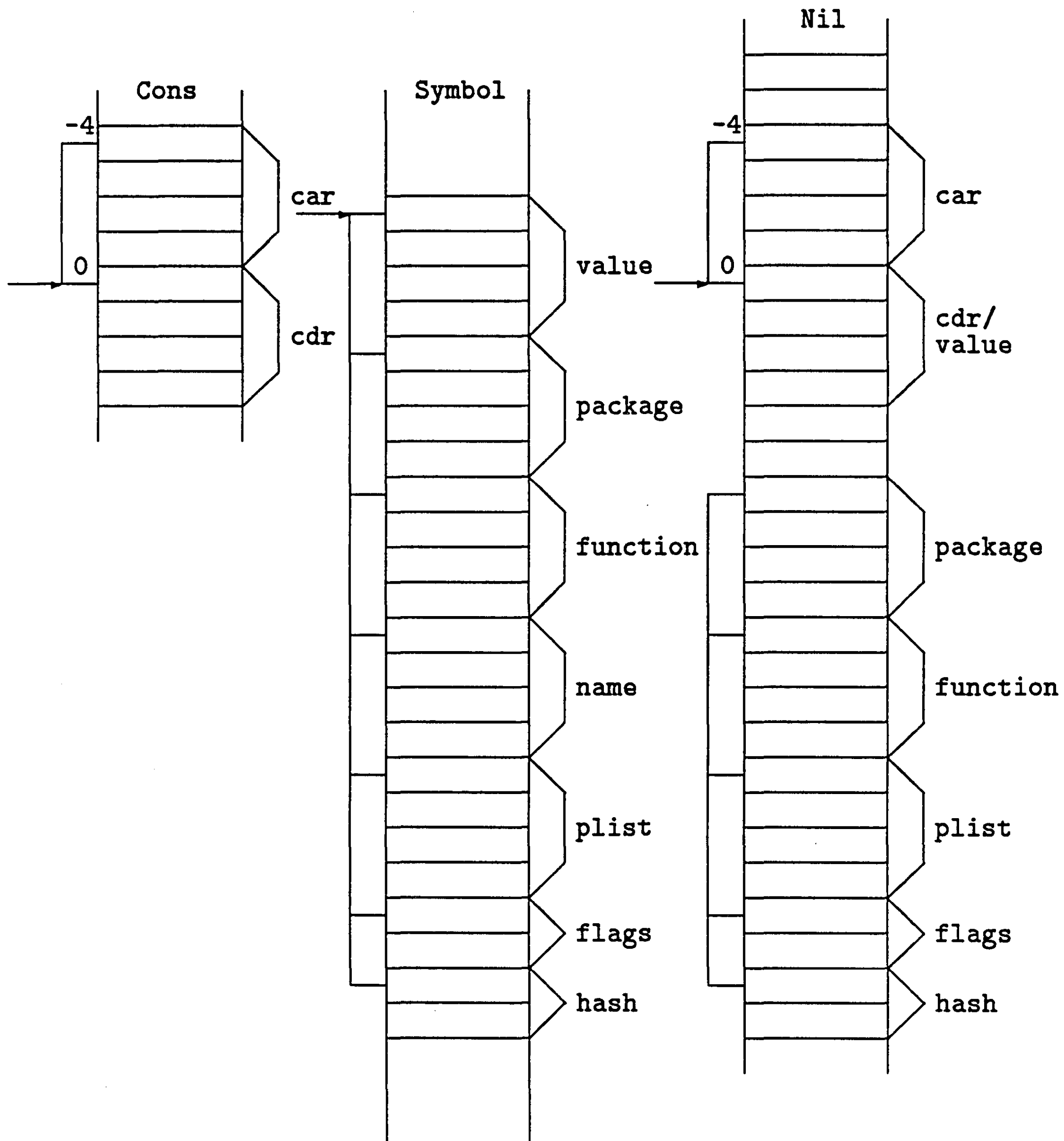

Figure 2.15. Representation of NIL in ExCL

vectors, various types of function objects, I/O ports, and environments are all allocated into pages. Fixnums are 15-bit 2s-complement; page table entries are reserved to flag fixnums and characters, although they need not actually point to distinct memory addresses.

Mark-and-sweep GC is performed by default; if this does not recover enough memory (because of fragmentation over pages), a compaction will be run also. The time needed is just a few seconds, even in the worst case.

2.1.42 mini-Scheme

Marc Feeley at the University of Montreal has written a small Scheme system for the 68000, using a native-code compiler written in Prolog [J. Dalton, personal communication].

The datatype representation was designed for efficiency; it is based on 32-bit pointers with 1–3 low tag bits, in the following assignment:

xx1 31-bit floats. (Also some unused bit patterns are assigned to `#t`, `#f`, `()`, and to characters.)

x10 Assorted objects (word *before* this address supplies the actual type).

100 Pair. (cdr before this address, car at it.)

000 29-bit fixnum.

2.1.43 XLISP

XLISP is a widely available public-domain microcomputer implementation. The language is a subset of Common Lisp and Scheme [47], and includes 9 types: list, symbol, integer, string, object, file pointer, float, and two kinds of builtin functions (subr/fsubr). It has no official description of internal structure, but the source code is a 7300-line C program. It has been ported to a large number of machines, ranging down to 16-bit microcomputers. This is at least partly because there is no compiler or other machine-specific optimizations.

(This description is for version 1.6.) Data representation is defined in terms of C structs. The most basic object is called a *node*, and always contains a `char`-sized *type* field, a `char`-sized *flags* field, and a union of structures, which should be no larger than two pointers. The flags field includes two mark bits for GC (two are needed because the marking algorithm is not recursive). Conses store the car first. `Nil` is 0, and car/cdr of `nil` is handled by explicit tests in those functions. Strings have a static/dynamic type, which is an int field in addition to a pointer to the string itself. String contents are malloced, and freed when the GC frees up the string node (this has the effect of relying on the malloc/free allocation system to handle all variable-size objects). Symbols have only property lists and value cells. The car of the property list is actually the string representing the symbol name, while the cdr is the property list proper. XLISP is like Scheme in not having function cells distinct from value cells. Objects have a class and a pointer to their data. The object data is actually a list long enough to hold all instance variables of the object. Classes are themselves objects, with 7 instance variables.

sometimes be used for type discrimination as well. The primary types are even/odd fixnums, "other" numbers, conses, symbols, procedures, "other" pointers, and "other" immediates. The fixnums are therefore 30 bits in length, and are therefore also valid word addresses in a 32-bit byte-addressed memory, which is useful for system building and debugging.

The "other" immediate types include short floats, characters, an object flagging unbound symbols, byte specifiers (not a first-class type of Common Lisp, but important [146, pp. 225-228]), and *header tags*, which appear in the headers of large objects such as arrays. Short floats have 16 bits of significance and 8 bits of exponent, while characters have 8 bits of code, 4 bits of "bits", and 12 bits for fonts.

Pointers are all offset by an amount appropriate to the tag, so that the first word of the object may be accessed without tag stripping. Cons cells do not have headers, so the cdr is stored at a 0 offset and the car at a 4 offset. All other allocated objects have header words which include both the type (a header tag) and a length if appropriate. Symbols are among the objects with a header word; instead of a length (the size of a symbol object being fixed), the header includes flags and a 16-bit cache for the value of **sxhash** on that symbol.[5] `Nil` is handled in basically the same way as for ExCL. The Common Lisp datatypes that do not have dedicated primary or secondary tags are implemented as ordinary structures.

Heap allocation is straightforward, with an extra wrinkle: the pointer to the next available heap location is already tagged as a cons, to speed up consing.

2.1.40 Lisp/370 and Lisp/VM

Lisp/370 was built during the 70s by IBM. Its internals were briefly described some years ago by White [164]. LISP/VM is the successor to Lisp/370 [5]. Objects are 32 bit tagged pointers, with 8 bits of type tag, and 24 bits of address.

2.1.41 PC Scheme

PC Scheme is a commercial implementation of Scheme for the IBM PC, developed by Texas Instruments [16]. It is based on a byte-code emulator at the approximate level of Pascal P-code, but unlike P-code, it is based on 64 32-bit registers instead of a stack. There is no method for evaluation, but the compiler is fast enough to be unobtrusive in use.

Objects are three bytes, divided into one byte for a page number and two bytes for a page displacement. Pages are variable in size, defaulting to around 4K bytes. Pointer chasing involves indexing a page table and adding the address to the page displacement (thus sacrificing some access speed, but getting some VM-like flexibility in a machine with no VM). An additional table stores the type of objects in each page. Lists, bignums, double-precision floats, symbols, strings,

[5]The **sxhash** function is potentially expensive to compute, so it is worthwhile to cache its results.

2.1.44 VT-LISP

VT-LISP is another microcomputer LISP, considerably smaller and simpler than XLISP. VT-LISP has been described by its authors Bev and Bill Thompson [157]. It is a pure LISP, with less than 20 primitives and only four special forms. The only types of objects are conses, symbols (limited in length to 80 characters), and numbers. VT-LISP is written in highly portable Pascal; at 1300 heavily-commented lines of source, it is one of the smallest working Lisp systems available.

The basic unit of representation is called a *node*, and is defined as a Pascal structure consisting of an enumerated type discriminating four types (the three mentioned above, plus a `free_node` object type), a mark flag declared to be `boolean`, and the data, which includes two pointers (conses), a `real` (numbers), 80 bytes (symbols), or a pointer to a block of free nodes. The pointer to a node is a 32-bit quantity, divided into a 16-bit segment number and a 16-bit segment offset (this is partly to deal with the 8086 architecture). Pascal's NIL is also VT-LISP's NIL. Symbols are not interned. Instead, they are compared by string comparisons always, since each re-read of the same symbol will cause a new symbol to be created.

The GC method is mark-and-sweep, using the mark flag attached to each object. Free space is maintained as a linked list of blocks of nodes, since the variable-length strings used with symbols share space with regular nodes, and fragmentation can be a problem.

2.1.45 CScheme

CScheme is a C-based system developed at MIT by the original Scheme group. The dialect is a large superset of Standard Scheme, with many builtin functions ranging from process handling to graphics interfaces. The implementation is based on a virtual machine interpreter, but is extremely large and complicated; the VM is over 50,000 lines of fairly portable C.

The virtual machine defines objects to have at least a 6-bit type tag, distinguishing no less than 62 (!) types of objects. This is more types than in any other system described; the types appear to fall into three groups: normal data objects (basically those of Standard Scheme), internal data objects, and source code constructs (for instance, a "conditional" type). At present, the type field is defined to be 8 bits in the high end of the word, along with a 24-bit data field. Symbols are two-component structures (name and value). There is a 3- and 4-component structures called `HUNK3` and `HUNK4`, which are treated very much like list cells. Vector length and type may both be found in the first word of a vector. Floats are double precision.

2.1.46 GNU Emacs Lisp

The Emacs implementation developed by the Free Software Foundation [145] includes a full Lisp as its customization language. Although used in a rather special context, GNU Emacs Lisp is quite complete; it includes about 22 primitive types (many specific to editing, such as buffers and windows), a garbage collector, and a

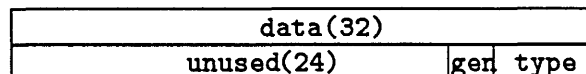

Figure 2.16. A Data Object in SPUR

compiler as well as interpreter. Like the rest of GNU Emacs, the Lisp is written in C.

The representation is expressed either as a union or as structures, depending on the settings of various compiletime flags. In either case, the representation uses 7-bit tags, 1 mark bit for GC, then a 24-bit data field. The tag/mark may be at either end of a word, whichever does not require offset addressing to examine (this is set at compiletime). Strings have 31-bit length fields in the first word—the sign bit is used as the mark. Vectors and symbols are unusual in that in addition to the usual fields, there is also a **next** field pointing to other objects of the same type, presumably to allow GC without moving objects around.

2.1.47 SPUR Lisp

SPUR Lisp is a Common Lisp system designed to run on the SPUR, a multiprocessing workstation system being developed at UC Berkeley. The SPUR CPU is a 40-bit RISC processor with a 32-bit address space. The 40-bit registers are actually divided into a 32-bit data pointer, a 6-bit type tag, and a 2-bit GC generation number (the type and generation bits together are the *typegen* field). The memory is byte-addressed.

SPUR Lisp internals have been described in a recent technical report [173]. SPUR Lisp was originally derived from Spice Lisp, and shares much of its original structure. On the other hand, the lowest-level representations diverge significantly, mostly because of the hardware support. Pointers in SPUR Lisp must be doubleword-aligned, with the first 32-bit word containing the data, and the next word containing 24 bits of unused space and the 8 bits of typegen information, as shown in Figure 2.16. Despite the longer data, there are only two immediate types: fixnums and characters, for which the most significant bit of the tag is 0, while all pointer types have a 1. Fixnums are normal 32-bit numbers, while characters consist of 8-bit code, 8-bit "bits", and 8-bit font fields. ([173] explains that although the 32-bit IEEE floats would seem to be logical candidates for immediate representation, SPUR cannot transfer directly from registers to the floating-point processor, so they have to be in memory anyway. However, pointer representation for floats also incurs allocation/deallocation overhead.) Tags for the allocated number types (floats, bignums, etc) are all assigned contiguously, although the existence of hardware type dispatching means this has no special advantages.

`Nil` is represented as a symbol, with all the same fields. car and cdr of `nil` address the same slots as a symbol's value and function. *I-vectors* store raw data,

and include bignums and strings as special cases. The header of an i-vector includes its *access type*, which indicates how many bits/element are stored (as an exponent of two; 3 = byte items, 5 = 32-bit items, etc). The header also includes a 4-bit subtype, and lengths in both 32-bit words and in number of elements. Strings are also terminated with a NUL character, for Unix and Sprite system compatibility. General arrays have many fields, with the contents stored separately (possibly with several arrays pointing to it), as in Spice Lisp.

Unlike Spice Lisp, the heap is only a limited region of memory. The GC algorithm is a generation collector of the sort first proposed by Lieberman and Hewitt [97]. There are four generations distinguished by the hardware bits. The heap divides into contiguous spaces, one for each generation, but they have no requirements on relative position or size.

2.1.48 LMI K-processor

The LMI K-processor was a next-generation Lisp machine project that fell victim to LMI's bankruptcy [J.R. Marshall, personal communication]. It was designed to run Zetalisp and Common Lisp, and exhibited extremely good performance.

The basic design of the processor is a derivation of the original Lisp Machine—the tag is in hardware and so forth. The basic object consists of a 6-bit tag field and 26-bit data/pointer field. Cons cells are even-word aligned, with the car first, and the cdr is accessed using special hardware that can OR in a 1 to the virtual address. Cdr-coding is no longer used. LMI's data indicated that only about 20% of memory is typically used for cons cells, and only 20% of those cells are in compacted form. Abandoning cdr-coding simplifies and speeds up both the hardware and software. `Nil` looks like a cons cell, followed by symbol structure components. Only `symbol-name` needs to check for `nil` as a special case. Fixnums are 24-bit rather than 26-bit objects, to accommodate the ALU hardware. Arrays are handled in a clever way that allows checking for bounds and array complexity (simple vector vs general array). The length of a simple vector is stored in the array header as a negative number, while the length of an n-dimensional or forwarded array is stored as a positive number.

The GC distinguishes ephemeral from permanent data objects, and must be able to scan the heap starting at any location. Cons cells, vectors, and code are stored in different areas.

2.1.49 UtiLisp

UtiLisp is a Common Lisp-like dialect built at the University of Tokyo [79,159]. Implementations have been in use since about 1981, but were more recently rewritten for 24-bit address machines (68000), and for 32-bit machines (VAX, 68010/020). The new one is dubbed UtiLisp32, and is written in LAP, which appears to be an abstract assembly language.

UtiLisp32 uses a limited form of the "space-tag equivalence" of Spice Lisp. The highest two bits of a 32-bit word are a tag field in which 11 designates fixnums, a

tag of 10 designates "other" objects, and the other two tags designate heap/stack objects. The only actual allocated memory is for heap objects. The heap itself consists of four areas that can grow into each other: compiled code, symbols, general objects, and cons cells. Cdr is stored before car. General objects start with a header word that gives a type, followed by a 32-bit length field (even for fixed-length objects like floats), followed by the data. Strings are zero-terminated as well.

Oddly enough, the two lowest-order bits of a fixnum must be 0, in order to be compatible with the allocater that always allocates on word boundaries.

2.1.50 A-Lisp

A-Lisp is a Common Lisp subset targeted to the Atari ST, presently under construction by Sandra Loosemore [99]. At this writing (summer 1988), it can run most of the Gabriel benchmarks, but is not in regular use. It is written in a combination of C and Lisp.

The data representation is a combination of BBOP and low tag bits. The two least-significant bits of a 32-bit word specify fixnums, short floats, characters, and all other types (which are all referenced through pointers). Fixnums get the 0 tag. Despite the available space, characters do not have bits or font attributes. Short floats consist of a 22-bit mantissa, 7-bit exponent, and a 1-bit sign, and are essentially in the Motorola format used by the C compiler. Pointers have a low tag of 2, and the objects are always aligned to be "off by 2" so that the tag need never be stripped when following pointers. Pages are 32K bytes each. Instead of using a separate type table, the type of a page is stored in its lowest address, which requires only a mask and indirect addressing, but no load of a page table address nor allocation of the table initially. Car is stored before cdr, although is asserted to have been a "random assignment." On the other hand, the function cell of a symbol is stored first to speed access, while the Common Lisp standard's permission to do anything with the function cell of `nil` is exploited by storing a self-pointer there, so car of `nil` needs no special treatment. Block-allocated objects such as vectors, strings, and structures have two-word header with a pointer to contents and a 32-bit length (which is not strictly necessary since object size is limited to 32K bytes). Packages are implemented as primitive datatypes, and are stored as single blocks of memory; two regions in the blocks serve as hash tables for internal and external symbols. Collisions in the hash table are resolved by chaining through links stored in special fields of the symbols.

The garbage collector is based on traditional compaction, but with the pages linked together, so that entire pages can be freed if possible.

2.1.51 SIOD

SIOD stands for Scheme in One Defun, written by G. Carrette as a smallest possible Scheme in Common Lisp. It was also translated into C, and in that form is comparable to VT-LISP in size [G.J. Carrette, personal communication]. The dialect is a subset of Standard Scheme.

The basic object is a C struct, consisting of a `short` GC mark, a `short` type, and a union of cons cell, flonum, symbol (with print name and value slots), subr (pointer to C function), and closure (with environment and code). The type field distinguishes these, with the additions of the tag 0 for `nil`, and 7 subtypes of subr (distinguishing different call/return protocols). The symbol names are C strings `malloc`ed and never recovered. `Nil` is represented as address 0 (or `NULL` in C).

GC is stop-and-copy, but with a twist; it can happen only between toplevel prompts, which means that the tracing process need not try to examine a C stack, but of course a too-complicated program will exhaust the heap and result in failure. SIOD is only intended for pedagogical purposes, so this is acceptable.

2.2 Purely Functional Languages

Purely functional languages, although related to (and inspired by) Lisp, have evolved along different pathways and have somewhat different characteristics. Since destructive operations are not supported, the implementor has more freedom to use alternate representations. In practice, this freedom seems to have been used more for parallelism than for high performance on single processors. In addition, the role of datatypes has been minimized, in some cases to the point that only small integers and lists are builtin, while all other structures must be constructed by the programmer. The net effect has been a dearth of data structure descriptions for purely functional languages. This section does include several descriptions of the FP language described by Backus [10]. (Probably the most available is one done by Baden at UC Berkeley [49], but this one is a rather simple program in Franz Lisp.)

2.2.1 IDRIL

IDRIL is a simple implementation of FP, with some facilities for utilizing hardware interrupts directly. It was written in Z80 assembly language, and partially described in [139].

There are three types of objects: names, numbers, and sequences. Internally, a 3-byte representation is used, with one byte as type tag, and two bytes as a pointer (the Z80 having a 16-bit address space). Names and numbers are actually quite similar, since a symbol table was never implemented. Sequences in FP are more like vectors than lists, and IDRIL signaled the end of a sequence with an all-zero object (which also represented the empty sequence—a mistake never discovered at the time).

Sequence structure is never shared, and of uniform size, so a free-list structure is used, and unused objects released explicitly (as a normal part of the primitives' operations). Since the released objects can potentially be large and complicated structures, they are not flattened into a list, but are instead connected to the free list, which becomes something more like a "free tree." When the cell to be allocated has two pointers, the second is pushed onto a stack. Thus allocation works from the top of a stack of trees; when a tree is entirely used up, the stack is popped.

2.2.2 Illinois FP

The Illinois FP interpreter is a recent implementation of FP done by Arch Robison at the University of Illinois [128]. It is notable for running both on single processors and several parallel machines. The interpreter is written entirely in C. The language is actually an extension of Backus' original dialect; new features include floating point numbers and strings, as well as a representation of functions as data.

The basic data object is a C `char` tag (which actually distinguishes 9 types), an `unsigned short` (usually 16-bit) reference count, and a union of the data fields for all other structures. Storage is allocated for all types of objects. If the reference count overflows (unlikely, but possible), then the object is copied (legitimate, since no destructive operations in FP). Sequences may be represented either as arrays or as lists, which is decided before building the interpreter. Strings are always represented as linked lists, with about 12 characters per cell (exact number depending on the machine).

2.3 Prolog

Prolog systems have not yet evolved as much as their Lisp counterparts. Most Prologs have only a few primitive types and emphasize basic compilation or interpretation rather than details of the runtime system. Data, program, and execution state are frequently intertwined in much the same way as in normal-order functional languages, greatly complicating the task of describing data structures. The most common implementation technique is known as *structure-sharing*, essentially a technique by which the structure of terms is distinguished from bindings, and dynamic storage allocation is minimized. However, it has some difficulties, and debate continues over the best methods for implementation. Unfortunately, most optimized Prolog systems have been commercialized, and data on internals are hard to come by.

2.3.1 C-Prolog

C-Prolog is a C-coded interpreter written at the University of Edinburgh [121], which has passed through several generations of rewriting. The dialect is "vanilla" Prolog, with only small integers, floats, symbols, and terms as data types. It uses separate areas for stacks and trails that are part of a structure-sharing interpreter, while objects as such are scattered throughout these areas, and are tagged with 1 or 3 bits in the low end of the word, as indicated in Figure 2.17. In order to avoid heap-allocation of floats, C-Prolog resorts to a highly questionable tactic: three bits of the significand is dropped, in order to make room for the tag. This is not documented, although no promises are made about the floating precision either, meaning that C-Prolog's floats are basically unusable.

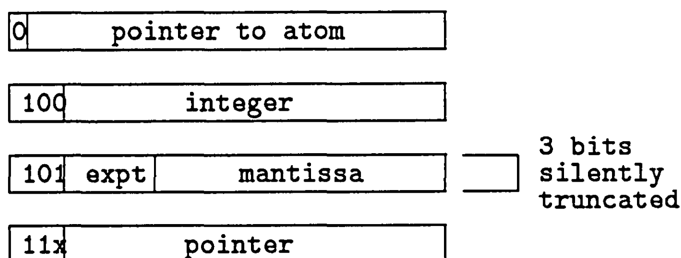

Figure 2.17. Data Representation in C-Prolog

2.3.2 SB-Prolog

Stony Brook Prolog was developed by D.S. Warren and several of his students. It features a high-quality compiler that compiles to a WAM that is then simulated with a C program. Details of data representation are buried in the source code [39].

The basic representation assumes 32-bit words, of which the three lowest bits constitute the tag. The two lowest bits distinguish free variables, constructions (non-atomic terms), numbers, and lists, while the next bit up distinguishes floats from integers. SB-Prolog uses its own 29-bit representation for floating point numbers.

2.4 Object-Oriented Languages

Object-oriented languages might seem to offer little interest relating to primitive datatypes. The use of an interactive and uniformly object-based system encourages (but does not require) uniformity of representation, and in fact that is what we usually see. An object cannot always be distinguished with a few bits of type tag, but may need a full-size pointer to a "class" object somewhere else in memory. However, uniformity tends to be expensive on the average, so there is a recent trend toward making more types of objects primitive, as a way to improve performance.

2.4.1 CLU

CLU is an object-oriented language developed at MIT [98]. The implementation described here is for PDP-10/20 machines [J.E.B. Moss, personal communication]; the VAX version seems to be similar, but the code is almost completely undocumented.

The basic 36-bit object reference has a 2-bit tag distinguishing positive and negative integers, pointers to objects, and object headers. Integers get the 0 and -1 tags. Pointers just use the lower 18 bits of the word. The lower halfword is also

used in variable-size objects for the size. The slots of objects and vectors follow immediately, as is the case for word/byte vectors and strings (characters packed five per word). These objects are discriminated by the 16-bit field (otherwise unused) in the first word. Arrays can also be more complicated, with an *array dope vector* including words for lower bound, size, and a pointer to the array data, possibly into the middle (the array data is itself a normal vector object).

Garbage collection (at least in later versions) is based on the Deutsch-Schorr-Waite algorithm [86, pp. 417-418].

2.4.2 Xerox Smalltalk-80

Smalltalk-80[6] is thoroughly described in the so-called "Blue Book" [58], which not only describes the language, but specifies a virtual machine on which the language must run. The virtual machine is essentially the Alto, a 16-bit machine with a 1M word address space (16-bit words). The virtual machine must provide a particular sort of representation for Smalltalk objects, so that the *Virtual Image* can run. The Virtual Image is best thought of a giant object code file containing the entire Smalltalk programming environment, from user interaction to pixel manipulations on the screen. It has been used by many different implementations, although some have altered the Virtual Image to conform to the representations in their virtual machine.

The original representation, based on the Alto, defines *object references* as 16-bit pointers into memory. The general object consists of a size word, a reference to its class, and 0 or more fields, all of which are 16 bits in size. The pointer to a general object has a 0 in the lowest-order bit position, while small integers have a 1 in the same place (the bit is essentially one of the Alto's condition codes). There are two kinds of collections of small integers, one for 8-bit positive values, and another for 16-bit positive values.

An object reference does not point to the object directly. Instead, it points to a large *object table*, which is just a vector of 32-bit entries, each of which contains a 20-bit pointer to the actual object, an 8-bit reference count, and some miscellaneous bits. (see Figure 2.18). The rationale for this is that the available memory is more than can be addressed by 16 bits, and that objects average about 10 words in size. Therefore, the object table effectively increases the address space by a factor of 10, while losing 65K words of memory. The available real memory is up to one megaword, so the loss of 6% of storage is offset by the 20% savings due to the use of 16 instead of 20-bit pointers.

For storage reclamation, Smalltalk-80 includes an 8-bit reference count in each object table entry, which is considered to have overflowed at 128 references (i.e. the high-order bit flags the overflow). There is also a marking GC that works by zeroing all reference counts in the object table, then incrementing as it traces from root objects (the current process and the global dictionary).

---

[6]Smalltalk-80 is a trademark of Xerox Corporation.

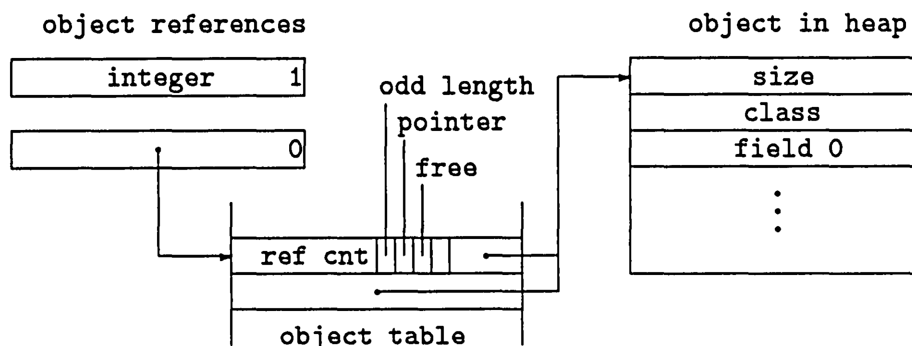

Figure 2.18. Objects in Smalltalk-80

A subsequent experiment in increasing the available memory while retaining short pointers resulted in the LOOM (Large Object-Oriented Memory) extension to Smalltalk-80 [77]. LOOM is a 4-gigaword memory (16-bit words). Each object includes a 32-bit field specifying its address in secondary memory. Its other fields are all 16-bit pointers, but each such pointer can be identified as a *leaf*, meaning that it really exists in secondary and not primary memory. There are a number of complexities involving with mapping 16- and 32-bit pointers back and forth, but this is compensated for by the increased speed and decreased space required by 16-bit pointers. In essence, LOOM is a Smalltalk-adapted combination of cache and virtual memory system.

2.4.3 Tektronix 4406 Smalltalk

This implementation built on Tektronix's experience with their first implementations, which were extremely close to the Xerox virtual machine but very slow as well [90]. The 4406 is a 68020-based machine, and its Smalltalk is intended to be fast enough to be usable for development and delivery of large applications [31].

4406 Smalltalk still uses the Smalltalk-80 Virtual Image, does away with the object table, and makes object references be 32-bits, of which one bit is an integer/pointer tag. 31-bit integers make the old "large integer" code in Smalltalk-80 unnecessary. Each object consists of at least three 32-bit words, as illustrated in Figure 2.19. Although the object table is gone, the garbage collector in the Virtual Image remains unchanged, so a 16-bit hash field is still necessary. Indexable objects such as arrays were broken up into the object proper and into a *remote object* with the contents of the arrays. Compiled methods are somewhat complicated, each compiled method consisting of three separate objects: a CompiledMethod object with pointers to various places, a LiteralArray object containing pointers to all objects referenced by the method, and a BytecodeArray with the code itself. Statistics are that the new system is larger, but also much faster (25 to 50 percent).

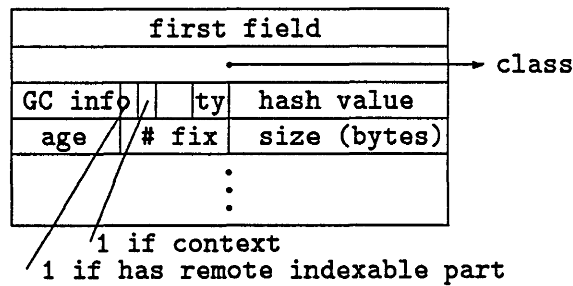

Figure 2.19. Object Header in 4406 Smalltalk

2.4.4 Swamp

Swamp is primarily a hardware implementation of Smalltalk-80 [96]. Like other recent implementations, it uses a 32-bit object pointer and no object table, with two high tag bits (0, -1 for small integers, to get a 30-bit range), and separate tags for general objects and *contexts* (stack frames). For these last two types, 3 extra bits are used for a generation number that is examined by a generation-scavenging garbage collector. Object headers are three words: object class, size of object, and a 16-bit hash value (represented as a small integer).

2.4.5 ConcurrentSmalltalk

ConcurrentSmalltalk is a Japanese development extending Smalltalk-80 for concurrent programming [170]. Objects are equated with processes. Although objects can work concurrently, there is no provision for sharing object memory in a parallel environment.

The virtual machine uses 32-bit OOPs (object pointers), with a 1-bit tag for small integers. Pointers point to an object table, since this facilitates forwarding and GC compaction. The general object consists of five 16-bit words, the first of which contains no less than 16 different flags, including everything from a mark bit, and flags for different types of primitive objects (blocks, contexts, methods, etc). This is essentially a bit-encoding of types. The remaining fields are a 16-bit reference count, a size for the fixed-length part, an overall size in bytes, and a pointer to the object's class. GC is based on reference counting, backed up by mark-and-sweep.

2.4.6 Little Smalltalk

Little Smalltalk is a smaller and more portable version of standard Smalltalk-80 written in C. Chapter 12 of Budd's description [26] distinguishes general objects

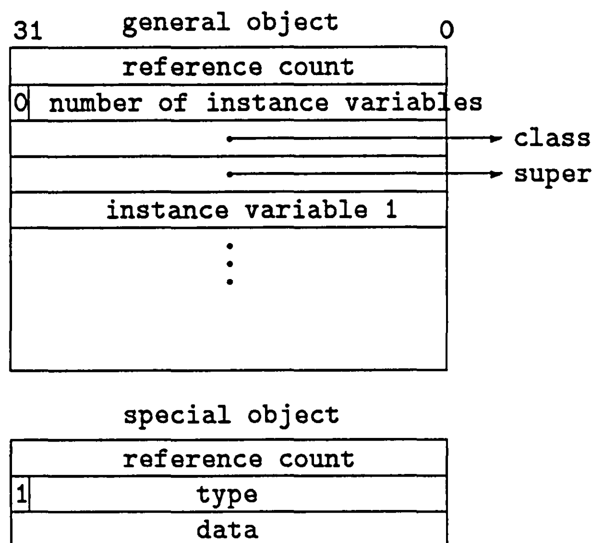

Figure 2.20. Objects in Little Smalltalk

from *special objects*, which are the primitive datatypes. General objects have a straightforward representation as sequences of words in the format of Figure 2.20.

Special objects are distinguished for the same reason that integers are distinguished in Smalltalk-80; space efficiency. Little Smalltalk is very comprehensive on this point, using special representations for these types of objects: `Block`, `ByteArray`, `Char`, `Class`, `File`, `Float`, `Integer`, `Interpreter`, `Process`, `String`, and `Symbol`. The representations for each of these types are quite straightforward, and usually not particularly compact. For example, a single character will occupy an entire word.

2.4.7 BrouHaHa

BrouHaHa [108] is, like Little Smalltalk, an implementation written in C to run on a variety of machines. Some special optimizations are done on the assembly language output by the C compiler.

Like the other Smalltalks on current hardware, Brouhaha uses 32-bit pointers, with a tag bit in the upper end of the word, with a 1 for integers and a 0 for pointers. The object table is retained, but the entries are 64 bits long, with an 8-bit reference count, 24 bits for a pointer to the class and 32 for the object body itself. The object itself has a 32-bit header consisting of one byte for flags and a 24-bit size field, followed by the instance variables.

2.4.8 UMass Smalltalk

UMass Smalltalk is a new implementation of Smalltalk-80 intended as a basis for experiments in distributed databases [J.E.B. Moss, personal communication]. It is written in C, and intended for 32-bit machines like the VAX.

The basic representation is a 32-bit word with 2–6 tag bits in the low end of a word. The following assignments are used:

xxxx00 Pointer to an object table entry.

xxxx11 Small integer.

xx0001 Small point. X and Y coordinates are each 14 bits.

xx1101 Small character.

000101 Nil.

110101 True.

010101 False.

All object references go indirectly through the object table. An object table entry consists of 4 words, where the first is the address of the object data proper, the second is a pointer to a class object, the third is the size, and the fourth contains various smaller fields, including 16 bits of GC info, and a count of the "fixed" instance variables of the object.

2.5 SNOBOL4

SNOBOL4, though chiefly known as a string-processing language, is notable for several other reasons: it was among the earliest of languages to employ pattern-matching and backtracking, to provide user-defined datatypes, and to include a wide variety of primitive types together with polymorphic operations on them. As such, its data representations are of great interest.

2.5.1 The Macro Implementation

Perhaps the best-described SNOBOL4 implementation is one built using a macro language, described in Griswold's 1972 book [66]. The macro language (called SIL) encapsulates most of the representation details described in Chapter 5. The basic data object is called a *descriptor*, and consists of a *T field* (tag), an *F field* (flag bits), and a *V field* (data). The descriptor object appears in several different contexts, and sometimes the T field holds normal data. Because of this, the T field must be larger than the few bits necessary for a dozen builtin types. For instance, it must also be large enough to encode all user-defined types. The Snobol4 system uses six flags, so the F field must be at least six bits, and the V field must be large enough for the largest integer allowed in the implementation.

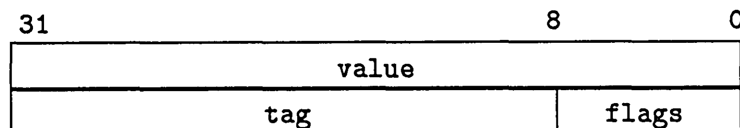

Figure 2.21. Representation in SNOBOL4

The descriptor for the IBM 360 is a total of 8 bytes in size, in which the first 4 bytes (a word) are the value field, and where the low 3 bytes of the second word are the tag field. The F field is 8 bits and thus fits neatly into the remaining byte, as illustrated in Figure 2.21.

The CDC 6000 word, on the other hand, is 60 bits long, and all the descriptor fields fit comfortably into it, providing 30 or 36 bits for a value (the 36-bit length is only used for floats), 6 bits for the F field, and 18 bits for the tag.

Storage regeneration (reclamation) is basically a mark-and-sweep garbage collector (although for some reason it is asserted not to be). The root for marking is a list of known blocks called *basic blocks* which point to all other blocks of descriptors. The mark bit is in the F field of a descriptor. The sweep process is straightforward compaction.

2.6 Icon

Icon is a recent language with roots in SNOBOL and SL/5. As with SNOBOL, there is a book describing the details of its implementation [67]. The version described therein is in C, and thus achieves a modicum of portability. The language is in many ways a modernized Snobol. Its set of data types includes strings, csets (character sets), integers, files, procedures, lists, sets, tables, records, and co-expressions, as well as additional types used internally only.

The C implementation assumes machines with 16- or 32-bit words and pointers at least as large as integers. The fundamental object is called a *descriptor*, and consists of two words, the *d-word* and the *v-word*, roughly corresponding to type tag and value, respectively. This setup supports strings specially by using the MSB of the d-word to distinguish strings from non-strings. The remainder of the string's d-word is its length, while the v-word points to the string's contents. For all other types, the remainder of the d-word is a type code (thus wasting some space, since there are only a few types of objects to distinguish).

Integers reside in the v-word or in a separate block of memory, depending on the relative sizes of the integer and the v-word. In general, the v-word of more complicated types is a pointer to a memory block, and the d-word then contains

an additional flag indicating this fact. The first word of the block is a repeat of the type code (necessary to assist the garbage collector with recognition).

Icon allocates strings in a separate area. They are not 0-terminated, since 0 is a valid Icon character; instead, the combination of length and pointer in the string descriptor suffices to identify the string uniquely. Since Icon does not support destructive operations on strings (modification by assignment works by modifying a copy), they can share as much storage as possible. In particular, substrings always share with the original string, and concatenation onto the last string in the string area does not copy that string.

Lists are actually doubly linked queues of vectors. An individual list element is actually a block containing arbitrarily many list elements in sequential order. The block may contain unused elements that are not counted as part of the list. This apparently bizarre representation is intended to optimize various common combinations of operations, such as arbitrary element access, additions to and deletions from a list, and append operations.

Sets are represented as hash tables (with 13 or 37 slots) whose entries point to 6-word *set-element* blocks, each of which includes a 2-word header (including the hash value), a 2-word member, and a 2-word pointer to another set element block. Tables are quite similar, but the table-element blocks are 8 words, so as to accommodate the two components of a table entry.

Storage management is based on the observation that many Icon programs never need to do GC. Allocated storage consists of a static region, a string region, and a block region. The static region contains only co-expression blocks, while the string region contains only characters. Since blocks vary in size, Icon uses a free block region pointer, allocating by incrementing a pointer; the string regions is handled in this way also. GC is mark-and-sweep, but since string space has no marks, there is a separate list of address pairs of string space that is in use. During the sweep phase, this list is sorted, then the characters can be compacted. Compaction in the block region has no unusual characteristics. An interesting feature of Icon storage management is that a program can reserve space ahead of time, triggering GC if necessary (one reason advanced for this is that untyped pointers may be present sometimes during execution).

2.7 APL

As SNOBOL is built around the idea of strings, so APL is built around the idea of arrays. The contents of an array must always be numbers (with the exception of Q'Nial; see below), but the array shape and size can be changed at any time. This possibility requires runtime representation, but although many APL implementations have been built, data on internal structure is scarce.

2.7.1 Purdue/Unix APL

The original Unix V6 APL was originally written by Ken Thompson. It has since passed through many hands, and is currently maintained by Purdue. It is

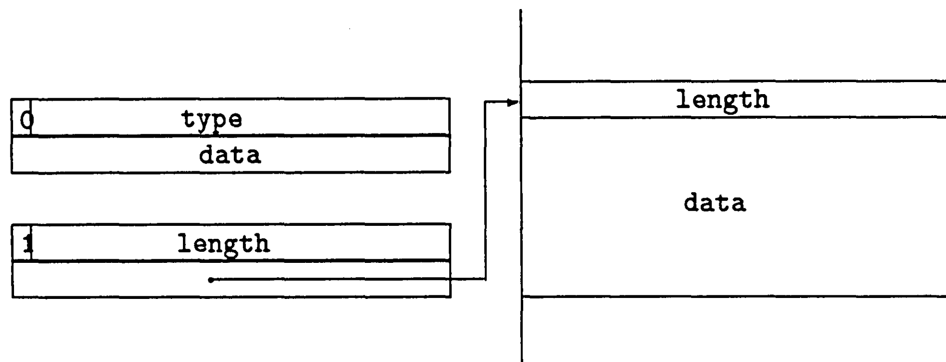

Figure 2.22. Representation in Icon Version 6.2

basically an interpreter, but includes a simple compiler that encodes the program more compactly. The system is written entirely in C.

The basic data structure is called an *item*, and consists of a type, rank (for arrays), size, index, pointer to data, and an array of dimensions. A limit of 8 dimensions is wired in. 16 types are defined. Allocation is dynamic and the array of dimensions varies in size with the declared rank, and contiguous with the contents at the end of the memory block.

2.7.2 Q'Nial

The Nested Interactive Array Language or Nial is a dialect of APL that allows array elements to be themselves arrays, while ordinary APL restricts elements to be atomic (numbers usually) [76]. Q'Nial is a portable C-coded implementation [75].

To obtain uniformity in handling, numbers and characters are defined as *atomic arrays* containing one element each. All arrays have headers containing a type, memory management data, a flag distinguishing atomic from non-atomic arrays, and one indicating whether the array consists entirely of data or of pointers to other arrays. Non-atomic arrays also include a pointer to a *shape array*, which is a normal array supplying all the dimensions (and which itself has a shape array!). Shapes are shared whenever possible. (See Figure 2.23.)

Storage reclamation is based on reference counting, which works well because Nial storage tends to be more stack-like than heap-like in its behavior. The problem of circular references seems to be solved by "cheating" with the symbol table references.

2.8 Conventional Languages

"Conventional" languages are those which are generally less abstract and more machine-oriented. Although the issues of runtime systems are usually negligible,

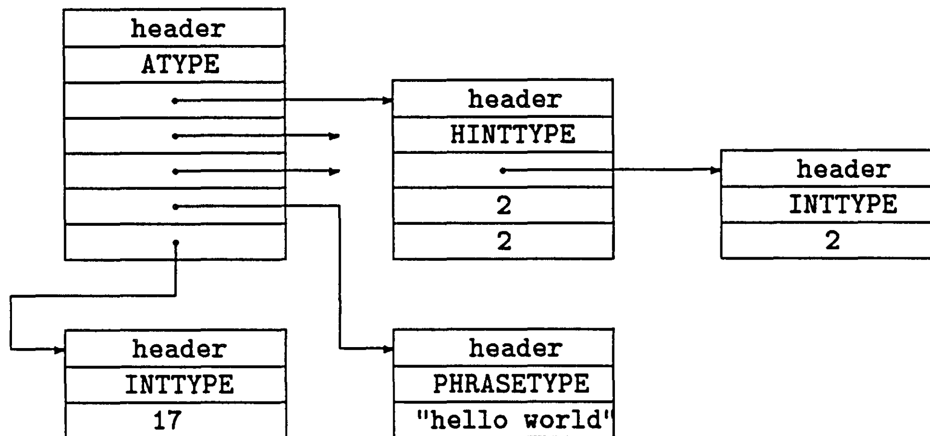

Figure 2.23. Q'Nial Array Representation

since conventional languages are at least partly designed to avoid any runtime overhead, there are at least two issues that have been discussed in the literature: representation of floating point numbers, and the representation of arrays, particularly those whose size can change dynamically.

The importance of floating point to scientific computation has meant that serious implementations are in hardware rather than software. Even so, there is much room for discussion on the merits of various representations. The IEEE format [118] is a specification both of behavior and of representation, but it is not universal, and many manufacturers have adopted their own form of floating point numbers. For a language system, the choice will usually be that dictated by the underlying machine, but for instance, many C compilers offer flags to choose the floating point representation to be used.

The run-time representation of arrays is a favorite topic of compiler books, such as Waite and Goos [160]. The choices for array storage (if not mandated by the language) include pointer-based or indexed addressing, and indexed addressing may be either in column-major or row-major order. An additional trick for indexed addressing is to allocate row/columns by powers of two, to avoid multiplication during indexing. The use of array size descriptors has already been mentioned for dynamically-sizeable arrays. Most conventional languages (Fortran, C) use indexed addressing with no tricks, to save storage and simplify inter-language interfaces.

Variable-size arrays, such as those encountered in PL/I, Algol (60 and 68), and Ada, require some runtime information to be maintained. The structures are referred to as *dope vectors* in this context [61], although they are not really different from typed structures in any of the languages considered so far. Dope vectors can usually be stack-allocated, and are known to describe arrays, so the structure is

rather simple; just a short vector of upper and lower bounds for each dimension, along with the number of dimensions and the base address of the array.

2.8.1 Ada

Although Ada was originally intended as a conventional language for writing embedded programs such as those in missile guidance systems, it ultimately ended up as a large language that requires significant runtime support [3]. Several features are defined in such a way that such support may be necessary: constrained subtypes, dynamic arrays within records, constrained heap objects, constraints on subprogram parameters, and the composition of record discriminants. Ada also allows (but does not require) automatic garbage collection, but few implementations actually provide it.

The traditional method uses dope vectors, as in Algol or PL/I. Hisgen *et al.* applied the technique to structures in Ada [72]. There, the runtime objects expressing type information are called *type descriptors*. There are several kinds, distinguished by a tag, and ranging up to a dozen words in size. Type descriptors are not typically shared by variables, even for variables that are all declared to be of the same type.

A recent paper by van Katwijk [158] describes the *doublet* model for Ada objects. Doublets are pairs of pointers, one to the object's storage, and one to an object descriptor. This is applied uniformly, so for instance even integers will be represented by doublets. This could be very inefficient, but the compiler does considerable analysis to eliminate nearly all actual doublet operations, and the object descriptors are shared when possible, as will often be the case when a number of objects are declared to be of the same type.

2.9 Summary

This chapter describes an utterly confusing mix of different languages and implementations. The confusion reflects to some extent what the thoughtful implementor is faced with when designing. A few patterns do emerge from the chaos.

First, there are very few major surprises among the systems, either very obviously bad or very obviously good designs. Clever designs are more often seen in heavily used systems, although no one can say whether the usage is in response to better designs yielding better performance, or the better designs were prompted by the expectation of heavy usage. Only a handful of released implementations have ever made a significant representation change after release; it would appear to be the case that the initial design choices are also the permanent choices. It is impossible to say what the real reasons might be; perhaps the difficulty of making any changes, perhaps the lack of high-quality performance data to direct the changes, or maybe only lack of interest.

Tags for type discrimination are clear winners over all systems, probably because they require fewer assumptions about the global structure of memory. BBOP is second, while full separate spaces are rare.

The primary focus of cleverness was on space usage before 1980, and on speed thereafter, which coincides with the general availability of large address spaces and

large real memories. In general, techniques have evolved to fit the hardware, with any delays attributable to software lifecycles (which appears to be about 5-10 years for these language systems; few have remained in use longer than 10 years).

Some implementation techniques never caught on elsewhere, such as the type-per-bit encoding of MacLISP. But this is uncommon; especially more recently, there has evolved some semblance of a consensus on the range of techniques and on the vocabulary to be used in talking about those techniques.

Representation "puns" are the highlights of implementation designs. The term is quite apt, since the same information is being interpreted in more than one way. Although it is not always clear that they are advantageous, a number of puns are well-known; others have only appeared in special situations. The most common puns should be familiar to any experienced implementor:

- Part or all of a tag is also part of a number.
- Tags allocated contiguously can also be handled as small integers.
- A tag in high-order bits is also the high-order part of an address.
- A tag in low-order bits is also an offset into an object.
- Addresses into unused areas of memory are also integers.
- Lisp `nil` is designed to appear both as a symbol and as a list cell.

The ostensible justifications for puns are evenly split between desires to optimize speed or space. Speed optimizations generally involve the elimination of one or a few machine instructions, while space optimizations count savings in individual bits. Some puns improve both space and time.

Throughout the designs there is an underlying tension between uniformity and special cases. Special-casing for more than two cases is generally only seen for numeric representations (both integer and floating-point) in Lisp systems.

Measurements of different schemes were never done, or perhaps never reported; some implementors have verbally asserted that tests were made, but these tests were either gedanken experiments, or unrepeatable once the implementation had been released.

This survey is merely a start at collecting and reporting basic information; a number of additional systems are known to exist, but descriptions are missing or incomplete. I plan to continue expanding coverage, and to work out methods for comparing the characteristics of different implementations.

CHAPTER 3

DESCRIPTIVE FORMALISMS

> ... the [Glass Bead] Game was so far developed that it was capable of expressing mathematical processes by special symbols and abbreviations. The players, mutually abstracting these processes, threw these abstract formulas at each other, displaying the sequences and possibilities of their science. This mathematical and astronomical game of formulas required great attentiveness, keenness, and concentration. Among mathematicians, ..., the reputation of being a good Glass Bead Game player meant a great deal; it was equivalent to being a very good mathematician.
>
> H. Hesse, *Magister Ludi (The Glass Bead Game)* (1943)

Although the implementations of the previous chapter span a wide variety of languages and machines, we can see a few patterns in the way that data is created, operated on, and destroyed. The most universal pattern is a notion of data objects being operated on by programs. Data objects are potentially unbounded in number and size and lifetime. Although real systems impose limits, programs rarely check those limits continually and explicitly. The objects almost always come into existence via explicit requests, and may or may not be discarded explicitly.

Data objects fall into several different classes. There are atomic objects such as bits—it is not possible to subdivide a bit. On the other hand, a "structure" has components each of which is itself a data object. Then there are arrays and array-like objects which include a large number of similar objects as components. More elaborate objects will also have constraints on their components, such as the requirement that rational numbers have nonzero denominators. Objects may also be identified with each other—again in the case of rationals, they are considered the same if they can be reduced to identical lowest terms.

3.1 Formal Definition of Types

The theory of types is one of the most intensively studied areas in programming language theory; many different models have been proposed [28,78]. No universally satisfactory theory has emerged, although the concept of abstract data types (ADTs) is widely accepted and has found its way into the standard curriculum.

Methods for the definition of ADTs range from the essentially syntactic forms of Modula-2 and Ada, to the highly mathematical approach embodied in multi-sorted algebras.

Purely syntactic approaches are practical for conventional programming, but they are not adequate for complete specification of a type. At the other end, the axiomatic definition of multi-sorted algebras introduces a great deal of uncertainty about the adequacy of a specification. Not only is it possible that an arbitrary set of axioms has no model, but it is not possible to determine this by analyzing the axioms—in other words, consistency is undecidable in general (although some techniques such as Knuth-Bendix completion can be used in special cases [73]).

As a way to resolve this problem, restrictions on axioms or on the type definitions have been proposed. Domain theory [135] is a function-based mechanism of great importance to proofs, but definitions of interesting types are complicated, being built from combinations of higher-order functions, and undecidability of axioms has been traded for undecidability of function equivalence. Cartwright [29] has advocated the use of a constructive type theory based on set operations like union and Cartesian product, an approach which provides the basis for the formalism used here.

The basic idea for this dissertation's formalism is to use algebraic type theory, but to select certain sets of axioms known to define types of interest. For instance, ordinary queues are defined by a particular set of axioms. This set of axioms could be represented by an *axiom schema* named `queue` that might be thought of a sort of "macro" that expands into the complete set of axioms. To handle queues with different types of elements or different names for the operations, the axiom schema could be parametrized, perhaps written `(queue integer cons-queue insert)`, where `integer` is the name of the element type, and `cons-queue` and `insert` the name of operations.

The point of schemas is not only to abstract away from individual axioms, but to provide types known to have finite models. This simplifies both the specification and the automated designer, and eliminates any question about the consistency or completeness of the axioms. The available schemas cover most types of interest, including ranges of numbers, structures, disjoint sums of types, and vectors.

Formally, abstract data types will be defined using Lisp syntax, basically consisting of a name attached to a schema:

`(defadt` *name schema* { *function* | *axiom* } `* ...)`

The *name* is the name of the type (a symbol), the *schema* is one of the schemas listed below, and both *function* and *axiom* are optional. Functions are usually auxiliary functions associated with the type, while axioms are sometimes useful as rewrite rules. (Note, however, that using axioms may result in inconsistency with the schema.)

We can assume some basic ADTs to exist already: in particular, the types `boolean` and `integer` have all the mathematical properties of those types as normally understood.

3.1.1 Standard Schemas

The standard set of schemas is based on what has appeared in the past as primitive datatypes in higher-level languages.

- Booleans and integers.
- Ranges of integers.
- Sums
- Structures
- Vectors

3.1.2 Booleans

The axioms for schema `booleans` need only ensure that there are exactly two distinct objects in the type:

```
(not (= t f))
(or (= x t) (= x f))
```

The model for this type can be any set of two distinct objects.

3.1.3 Integers

The schema `integers` designates the set of integers along with the usual assortment of numeric operations. Integers can rarely be used directly, because of their infinite extent, but they make useful building blocks. The schema includes names for a basic set of operations, such as `int+` for addition and `int-logior` for bitwise logical OR, while others such as `int<=` can be defined as nonprimitives.

This axiom schema would expand into a definition of integers based on Peano axioms, with the numeric operations appearing at various places in those axioms. This is a familiar process, so I will not repeat it here.

```
(defadt foo (integers int+ int- int* int/
                      int-logior int-logand int-lognot
                      int<)
  (define int<= (lambda (x y) (or (= x y) (int< x y))))
  (define int> (lambda (x y) (int< y x)))
  (define int>= (lambda (x y) (int<= y x)))
)
```

3.1.4 Ranges

Subranges of integers occur frequently in definitions, not only to to define numbers, but to index finite collections of other kinds of objects. The range type R is defined with a schema that includes lower and upper bounds m and n, respectively:

(defadt R (range m n))

This schema does not define any of its own functions, but it does inherit all the functions for integers, restricted suitably. The sole axiom just states that a object r in R is an integer falling into the given range (note that it is inclusive on lower but not the upper bound):

$$m \leq r < n$$

A variation that will prove handy is bit fields, which are essentially integers between 0 and $2^n - 1$ inclusive, but can be used to avoid writing the large powers of 2 that can appear frequently:

(bits n) $\equiv$ (range 0 2^n)

3.1.5 Sums

The sum schema defines a type that is a disjoint union between a number of subtypes. Objects in each subtype retain their individual identity. The only primitive operations defined by a sum are the predicates p_i distinguishing each type t_i:

(defadt S (sum (p_1 t_1) (p_2 t_2) ... (p_n t_n)))

The types t_i need not all be distinct, but two occurrences of the same type will be regarded as different subtypes of the sum. The only primitive functions defined are the type-testing predicates

$$p_i(x) : S \rightarrow \texttt{boolean}$$

while the axioms capture disjointness:

$$\forall x \in S \; \exists i, \; p_i(x)$$

An example of a sum is the type t of Common Lisp:

```
(defadt t
  (sum (numberp number)
       (symbolp symbol)
       (arrayp array)
       (characterp character)
       (consp cons)
       (null null)
       (packagep package)
       (hash-table-p hash-table)
       (random-state-p random-state)
       (readtablep readtable)
       (streamp stream)))
```

3.1.6 Structures

Structures are the same as records in many languages, `defstruct` in Lisp, and terms in Prolog. In the schema here, it is possible to have both mutable and immutable structures. In fact, we can generalize this to distinguish between mutable and immutable components of a single structure type. This is useful in the context of Common Lisp symbols, which are defined to have immutable components `symbol-name` and `symbol-package`, while the components `symbol-function`, `symbol-value`, and `symbol-plist` can be modified without affecting anything else. In the schema for structure type S, a component holding objects of type t_i, with an accessor function a_i, can also have an optional setting function s_i. The default creation function c takes all components as arguments; any other desired creation functions must be nonprimitives.

(defadt S (structure c (a_1 t_1 [s_1]) (a_2 t_2 [s_2]) ...)

The signatures are straightforward; note that the setter returns the modified structure, which simplifies handling later on:

$$c : t_1, t_2, \ldots \rightarrow S$$

$$a_i : S \rightarrow t_i$$

$$s_i : S, t_i \rightarrow S$$

The axioms need only state a sort of inverse relation between creating and destroying, and define how the setter works:

$$a_i(c(x_1, \ldots, x_n)) = x_i$$

$$a_i(s_i(c(x_1, \ldots, x_n), y)) = y$$

The Common Lisp symbol type offers a good example of a structure (recall that `t` means "any type" in Common Lisp):

```
(defadt symbol
  (structure basic-make-symbol
    (symbol-name string)
    (symbol-package package)
    (symbol-value t set-symbol-value)
    (symbol-function function set-symbol-function)
    (symbol-plist t set-symbol-plist))
  (define make-symbol
    (lambda (s p)
      (declare (string s) (package p))
      (basic-make-symbol s p '%unbound% '%undefined% nil))))
```

There is actually no standardized Common Lisp function to create a symbol with all of its slots filled in, so `basic-make-symbol` is a substitute, and the standardized function `make-symbol` is a nonprimitive filling in some default values. Likewise, the functions `set-symbol-value` and so forth are also not standardized; in a complete system, they will appear only in expansions of the `setf` macro.

3.1.7 Vectors

Vectors include both fixed- and varying-length sequences of objects, all of the same type. Like structures, vectors may also be mutable or immutable. For a vector type V, c is the creation function, m is the length, a is the element accessor, and t is the element type. Optional parts include the maximum length n (in which case m is a minimum length), the length function l, and a modifier function s. The syntax is such that fixed-length vector do not define a length function at all. If it is really needed, a length function could be added as a nonprimitive that returns the value of the constant m.

```
(defadt V (vector c (m [n l]) (a t [s]))
```

The schema defines from two to four functions, depending on the options. (In this definition, we assume mutable vectors, and a creator that takes only a length as argument).

$$c : \texttt{integers} \rightarrow V$$

$$a : V, \texttt{integers} \rightarrow t$$

$$l : V \rightarrow \texttt{integers}$$

$$s : V, \texttt{integers}, t \rightarrow V$$

Axioms:

$$l(c(n)) = n$$

$$a(s(c(n), i, x)) = x, \text{ if } 0 \leq i < n$$

Character strings constitute a familiar example of a vector. Here, we assume 8-bit characters:

```
(defadt string
  (vector make-string
          (0 1000 string-length)
          (schar character set-schar)))

(defadt character (range 0 255))
```

3.1.8 User-Defined Types

The type framework described does not allow for new types to be defined dynamically. Although full ADT languages allow for this possibility, the capability requires types to be (nearly) first-class objects, which is typically not the case for most higher-level languages. Most of the languages do have some kind of restricted type-definition facility.

Common Lisp has two type-definition facilities, with rather different characteristics. The first uses the `deftype` macro, and is fundamentally intended for the benefit of declarations. Types defined using `deftype` are all simple specializations of builtin types, and objects do not have distinct representations.

By contrast, the `defstruct` macro creates new types whose objects are distinct from all other types of objects. From a practical point of view, however, all `defstruct` types are quite restricted in their form; a fixed number of slots capable of storing anything (with the possible exception of typed slots, which a system need not support). Because of this, implementations have usually adopted a common representation, which is vector-like, with the slots matched with positions in the vector. This is not completely type-secure, since because `defstruct` is a macro, its expansion will usually reveal the implementation details.

The builtin part of structures is fairly simple, and it seems safe to assume that a similar approach could be taken for the user-defined types of Snobol and other languages as well. Thus, structures can always be defined as relatively short vectors:

```
(defadt structure
  (vector make-structure
          (0 1000 structure-length)
          (get-slot t set-slot)))
```

3.1.9 Examples

The classic example of rational numbers needs two axioms to distinguish true rationals from just any pair of integers.

```
(defadt Q
  (struct make-rational
    (numerator integers)
    (denominator integers))

  (not (equal 0 (denominator x)))

  (equal (* (numerator x) (denominator y))
         (* (numerator y) (denominator x))))
```

Simple S-expressions introduce the possibility of circular or recursive type definitions:

```
(defadt sexp (sum (atom small-ints)
                  (consp conses)))

(defadt small-ints (range 0 1000))

(defadt conses (struct cons (car sexp) (cdr sexp)))
```

Scheme is somewhat more complicated than S-expressions, but not by much:

```
;;; Scheme as defined in R3RS

(defadt scheme
  (sum (boolean? boolean)
       (null? empty-list)
       (pair? pair)
       (symbol? symbol)
       (number? number)
       (char? character)
       (string? string)
       (vector? vector)
       ;; procedures
       ;; i/o ports
))

(defadt boolean (set f t))

(defadt empty-list (set nil))

(defadt pair (structure cons))

(defadt symbol
  (structure string->symbol
             (string->symbol string)))

(defadt number
  (structure xxx
             (exactness exact-bit)
             (xxx bare-number)))

(defadt bare-number (sum (complex? complex))

(defadt complex (sum (real? real)))

(defadt real (sum (rational? rational)))

(defadt integer (sum (integer? integer)))

(defadt character (range 0 256))   ; not many promises

(defadt string
  (vector make-string
          (0 1000 string-length)
          (string-ref character string-set!)))
```

```
(defadt vector
  (vector make-vector
          (0 1000 vector-length)
          (vector-ref scheme vector-set!)))
```

The full definition of Common Lisp data is rather lengthy, so it has been relegated to Appendix A.

3.2 Formal Definition of Machines

Researchers have been interested in the modelling of hardware for a long time. Perhaps the best known of these formalisms is ISPS [12], which is a version of the ISP notation familiar to generations of students from Bell and Newell's classic text [19]. It is essentially a procedural language in which programs represent processors. It has been used in activities ranging from simulation of hardware to compiler retargeting in PQCC [94]. The APL derivative AHPL of Hill and Peterson [71] is similar, but somewhat less procedural, since it exploits the vector operations of APL. A number of retargetable compiler efforts have developed special-purpose descriptions, as for instance in the Bulldog compiler [44], Peep [83], and PO [38], among others. These descriptions are also procedural, but at a procedure/instruction level, and consist of only one or two statements apiece. Analysis at this scale is not only feasible, but even reasonably efficient.

Unfortunately, none of the aforementioned formalisms are directly usable, nor are their implementations. Existing systems have been designed for particular purposes, and the semantics vary widely—some languages are very precise about the forms of arithmetic and sequencing of operations, while others are too complicated to reason about effectively.

The machine will be modelled as yet another ADT, typically a structure with an extra function that models the machine's instructions. The structure is composed from all of the machine's storage areas, while the function is a next-state mapping from machine states to states—in other words, the total description of instructions. In practice, these will be partial specifications. Only those storage areas of direct relevance will be included, and the function will be expressed using rewrite rules covering the most useful instructions. This means that there is no way to talk about instructions doubling as data, or about self-modifying code, but this is not a currently accepted practice in any case.

Storage space will be modelled as vectors of integers. Bit arrays are also valid, but make the description of arithmetic operations much more complicated. It is important to characterize the actual sizes of things accurately. For example, Berkeley Unix virtual memory works in such a way that the full address space of a program is allocated on the swap device. This has the effect of limiting programs to the size of the device, which is probably less than the hardware's full address space. This will have an enormous impact on the space of possible designs. In

those cases where the amount of memory can vary, memory should be modelled as a varying-length vector. Many modern memories can be accessed in different ways, perhaps in 1-byte and 4-byte groups; this can be modelled by making one size part of the definition, and accesses of the other size into nonprimitives.

Storage that is not part of main memory is somewhat more complicated to handle, because some of it is irrelevant, and because most of it is used in special ways during execution. Registers for argument-passing and result-returning are an obvious example.

3.2.1 Examples

The VAX series has a rather simple storage structure. The definition of instructions has not been included.

```
(defadt vax (structure make-vax (r reg s-r) (m vaxmem s-m)))

(defadt reg (vector 16 vaxword))

(defadt vaxmem (vector #.(expt 2 30) byte))

(defadt vaxword (bits 32))

(defadt byte (bits 8))
```

The 68000 description includes a number of instruction descriptions, and may be found in Appendix A.

3.3 Representation of an Implementation

The obvious way to relate abstract types to machine types is to define a function. However, a mapping of types to machine words is not uniquely determined, since the same object may be represented in several ways. A cons cell, for instance, can equally likely be stored at address 0, or 1232, or 555, and it will still be the same cons cell. The inverse function maps machine configurations into abstract objects, and is a valid (partial) function. It is a partial function because some machine states may not represent any abstract object at all (i.e. garbage memory). Figure 3.1 illustrates a simple version of this.

D turns out to be insufficient; it only defines the relation of configurations, and says nothing about how those configurations are reached in the first place. Consider that the function defined above says nothing about storage allocation or reclamation, not even whether such operations exist at all. The set of models encompassed by the D above include not only implementations in which cons cells are shared, but in fact ones in which cells overlap, where the cdr of one cell is the car of the next!

The problem of design then divides into one of designing various Ds, then of designing the actual functions to be consistent with a particular D. This second

$$D : \texttt{register}, \texttt{memory} \rightarrow \texttt{sexp}$$

$$D(x, M) = x, \text{ where } x \in [0, 1000)$$

$$D(x + 2^{31}, M) = \texttt{cons}(D(M_x, M), D(M_{x+1}, M))$$

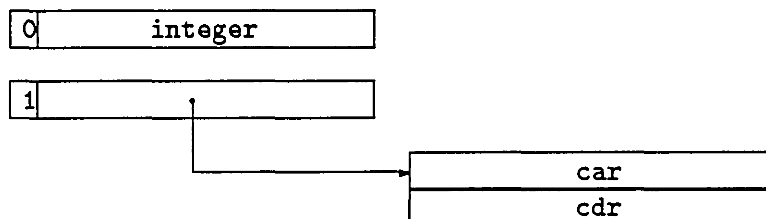

Figure 3.1. An Implementation Function

stage is extra complexity, and must among other things be able to compute inverses of D, which is impossible in general. Therefore, the implemented set of design rules synthesizes the primitives directly, expressing them in terms of low-level operations, which include the functions on integers mentioned earlier, as well as the primitives and nonprimitives of the machine ADT.

3.4 Pragmatics of Usage

The basic definition machinery developed so far is insufficient to explain the derivation of familiar implementations. It is not possible to specify lists in such a way that cdr-coding can appear to be desirable, nor is it possible to prevent a designer from giving up entirely, on the grounds that the machine is not big enough to accommodate any large data structure that a program might try to construct. The specification must include additional information commonly known as *pragmas.*

3.4.1 Finiteness

Like nearly all type systems, the formalism here allows for infinite objects of various kinds. However, a little thought should convince one that allowing the definition of potentially infinite objects is going to cause problems for implementations. Numbers are an easy example. Suppose the language specification insisted *all* integers be represented. Then a 32-bit, 64-bit, etc. representation is insufficient, since there are always integers whose representations require more than any given number of bits. Lisp implementors routinely implement arbitrary-precision integers (bignums), so as to allow very large numbers, but this only delays the problem,

rather than solving it. A one megabyte memory only has room for some eight million bits, which is still finite. A more subtle problem occurs in connection with list structure. Lists are basically interconnected small structures, but their definition is recursive. Thus, the amount of storage is again potentially unbounded, although each individual object has a small fixed size. This has a direct bearing on the design of a list cell, since both halves are indices to other list cells. For example, if a program references a million randomly interconnected cons cells, then both the car and cdr pointers must be at least 20 bits long.

It is not necessarily adequate to just say "as many as possible." In a language with several types of objects, this is an ambiguous phrase—should the finite memory be given over to more list cells or more numbers? An even division is not always desirable, since some programs will need more list cells, while others need more numbers at any given moment. The form of the representation is fixed ahead of time, so the designer must know how many of each type to plan for. For an example, consider s-expressions again and suppose that the mandate is to have "as many as possible." As many cells as possible means that the entire machine word should be an address, but then there would be no valid representation for a number. In a large language like Common Lisp, the dilemma is worse—how many of each of 30 types should be supported?

Therefore, any implementable ADT must have a set of specifications of sizes and numbers for all types mentioned.

3.4.2 Statistical Patterns

Some designs are motivated by statistical considerations not expressed in the type schemas. A familiar example is cdr-coding. In some well-known studies of real Lisp programs [33], Clark and Green found that most usages of cons cells were in long lists. In other words, the cdr of any cell was more likely to be another cons or NIL instead of an arbitrary object. Based on this, Bobrow and Clark [22] proposed an alternate representation of lists where each cons cell was a single pointer to the car and the cdr was assumed to be in the next word. This scheme has a number of obvious problems, for instance when destructive operations and various forms of list splicing occur, and it is impractical without the use of special hardware. Later versions allowed 3-8 bits for the cdr, which still gave a space savings while retaining some flexibility in list structure. The whole basis for this technique is a set of measurements of the average behavior of programs.

There are many kinds of statistical distributions for elements of a population. "Zipf's law" [172] is well-known and appears in many situations; it is a distribution where the nth most common member occurs with a frequency inversely proportional to n. Other distributions are exponential—the 2nd most common member occurs half as often as the most common one, and so forth (see [88, pp. 396-399]).

The typical situation in languages seems to be one in which small objects are far more common than large ones. For instance, small integers are much more common than large ones [137, pp. 50-51]. On the other hand, it also seems to be the case that positive integers were 200 times more common than negative ones,

so the distribution is somewhat lopsided. On the other hand, Common Lisp does use Universal Standard Time as the representation for time. UST is measured in seconds from January 1, 1900; at present (1988), this is something like 2781194496, which requires at least 32 bits to represent. In the usual distributions, this should be an extremely uncommon number, but in fact a program that did anything with dates and times would have many of these numbers, probably within a narrow range of values representing several weeks or months. Our distribution of sizes then has a notable "bump" in it, and the conscientious implementor might wonder whether the chosen representations will do unusually well or poorly on numbers within the "bump". Another likelihood is that heavily optimized programs will have clusters of values around each power of 2, used in bit-packed representations, for sizes that work out "right" when combined, and so forth. The distribution of values might have quite a few undulations in it, though at increasing separations.

Symbol names in Common Lisps provide another interesting case study. The average length of names is about 5 or 6 characters, but the distribution may be oddly shaped, as in the histogram of HP Common Lisp symbols in Figure 3.2; there are still many symbols up to 10–12 characters, and a very few long ones, up to 61 characters in length. This diagram clearly indicates that optimizing very short symbol names (1-3 characters) would be as unproductive as optimizing very long ones.

The obvious answer to all these considerations is to include the complete frequency distribution as part of the datatype description. The distributions are probability functions (area under curve < 1) whose independent variables may be any of several parameters, such as sizes of objects, numbers of distinct objects, and percentage of types of components. (Rosenschein and Katz [130] have introduced similar pragmas.) In practice, only the most general characteristics of a distribution are important, such as its maxima or the slope of a part of it. These are recorded with the ADTs along with axioms:

(**max-number** *n*) The maximum number of the type that will exist at any one time.

(**most-common** *slot type*) The most common subtype appearing in a given slot of a structure.

(**most** *value*) The particular member of a type that appears most often in programs.

A (**cluster** *value*) pragma to designate a value around which others tend to cluster would be useful, but this would require some sort of distance to be defined for the type (for integers, this could be the absolute value of the difference between two integers).

Recursive objects present a special problem. We must assume that each instance will be an individual object and specify the number of those, although alternate implementations might make the numbers meaningless.

Definitions for the size of the executable part of the language kernel would also be useful, as well as for the size of programs, but this is complex and will not be considered here, although such definitions would be essential in a full designer.

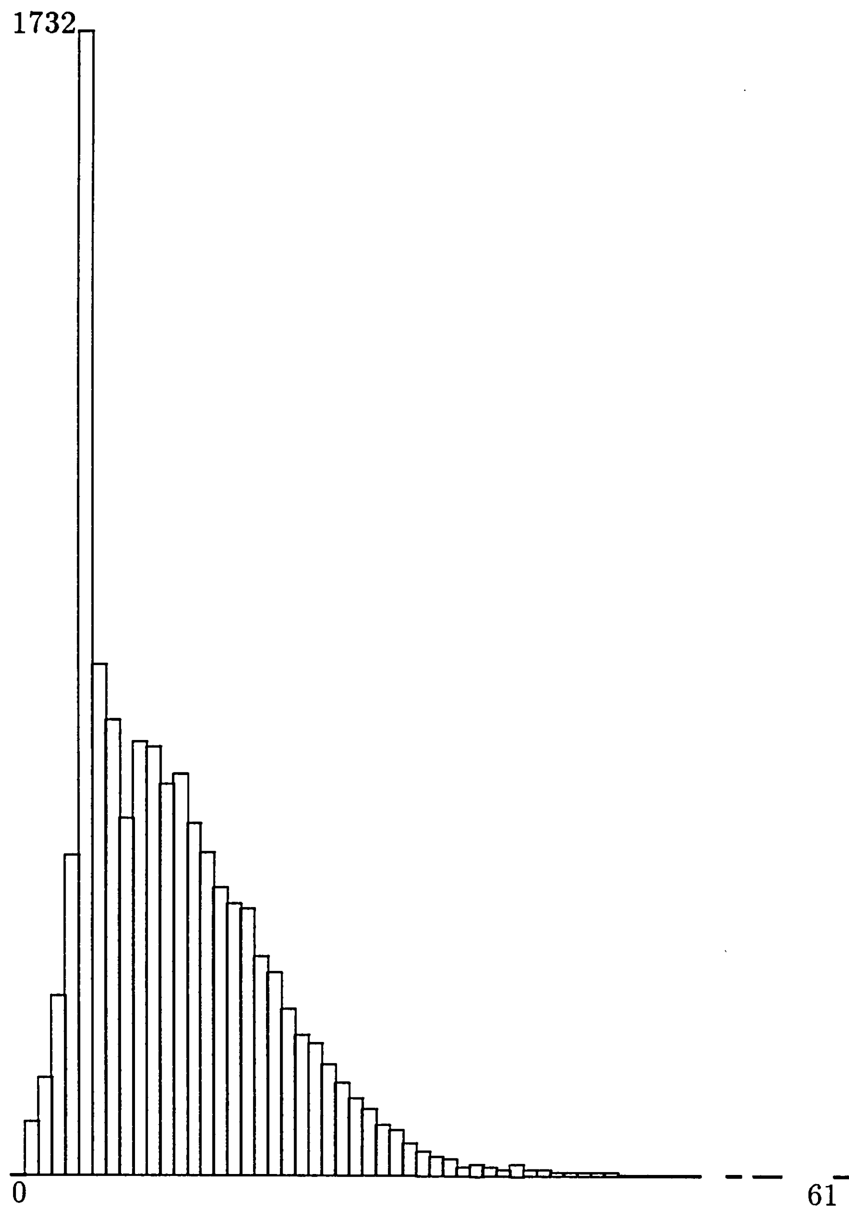

Figure 3.2. Symbol Name Lengths in HP Common Lisp

3.5 Pragmatic Limits in Standardized Languages

Although past standardization efforts were limited to conventional languages, both Common Lisp [146] and Scheme [126] have recently emerged as standards in the Lisp world. There are also efforts for a European Lisp standard [119], as well as for a Prolog standard. One unexpected consequence of the formalization in this chapter has been the discovery of several omissions in the standards' specification of datatypes.

Machine limitations are a perpetual problem for standards. Even in a relatively simple language like C, there is considerable variation in the sizes of `char`, `int`, and `long` integers; implementations have represented `int` integers with from 16 to 64 bits. The ANSI standard for C [7] does set some limits, such as requiring `int` to represent at least 16-bit numbers, and `long` to represent at least 32-bit numbers. Ada's specifications are moderately elaborate [3, ch. 13], and include specifications for all numerical limits.

The situation in Lisp dialects (and in other higher-level languages) is considerably more complicated, both because of the abstractness and because of the greater variety of objects. A Common Lisp program that needs a certain amount of memory may be perfectly correct according to the specification, and yet not run in a too-small system which is nevertheless a correct implementation.

It is theoretically valid for a CLCI (Common Lisp Conforming Implementation) to have room only for the symbols of the Lisp package (about 800). Only four packages are required, only one random state, only one readtable. It is not clear what the required range of integers might be; interpreted one way, the range need only be large to accommodate UST times (a 32-bit number).

This is not to say that no attention has been paid to the issue of sizes. Limits on arrays have been carefully specified; the rank of an array may be up to 7, each dimension must be allowed to be at least 1024, and the total size allowed must be at least 1024. Programs and programmers using arrays in CL can count on these limits in a CLCI. Similarly, the range of floating point numbers is also available.

A more generous way to interpret the Common Lisp standard is that objects can range up to the maximum size and number allowed by available memory. For example, the bottom of p. 13 of [146] says "Common Lisp in principle imposes no limit on the magnitude of an integer." Admitting this, however, means that a programmer will never know when a problem is too big until the program exhausts memory, and then that knowledge is only applicable to the one machine—the next one might not have as much memory, and a previously working program will fail. To put it simply, the goal of portability has not been achieved, and worse, the programmer can do nothing to make certain that the program is portable in the future.

Random states turn out to be a problem. Their size and character is not specified; however, the `random` function is defined to return numbers in any range specified by the argument, with no upper bound. This is misleading, because if the range is sufficiently large, then the random state will not be "random enough". In fact, if the random state $n < 2^k$ states (due to the periodicity of the generator),

then `random` will generate no more than n distinct numbers, no matter how big the bound is set. Therefore, the lack of specification of the size of the random state in CL means that portable code cannot rely on `random`, for *any* integer argument.

The Scheme standard has rather more serious holes; almost no promises are made about implementations' capabilities. In theory, the maximum size of a vector is allowed to be 0, and strings need never be longer than any name in the basic set of functions. The class of numbers upon which a program can depend is completely unspecified.

3.6 Summary

The mere act of formalization is useful, because it reveals unwarranted assumptions. This started out as a mechanical process of setting up formal structures for the purposes of implementation, but has ended up demanding the specification of things usually left vague:

- The range of sizes of objects.
- The probability distributions of objects.
- The true size of machine structures.
- The semantics of operations on the objects.

When we examine standardized languages, it is clear that although some limits have been set, others remain unspecified, and are therefore potential hazards to application programmers. Chapter 6 will include a set of recommendations for future revisions of the Scheme and Common Lisp standards.

CHAPTER 4

IMPLEMENTATION DESIGN RULES

> I have watched managers make the decision to use assembly language ... on the basis that "it has to be fast." Such a decision cost millions of dollars and was made with about 10 seconds deliberation.
>
> R.J. Rader, *Computing Reviews* (1983)

This chapter is essentially a compendium of rules relating to the design of primitive datatypes for languages. They derive from general knowledge about data structure design, from specific knowledge about existing language systems (such as those in Chapter 2), and from experience with automated designers of the sort described in the next chapter. Justification also derives from rational reconstruction of real designs, that is, by analysis of what rules are necessary to produce familiar implementations.

The formalization in the previous chapter largely reduces the task to one of ADT implementation; only a few rules are actually specific to languages and not programs in general. Many more rules assume a target ADT that resembles a machine. The rules themselves are rather vague; partly this is because of their expression in English, and partly because the bulk of real rules consists of mechanical processes constructing code fragments. In addition, many of the rules overlap or contradict each other; such situations result in multiple designs that cannot be decided on until coding or evaluation time.

4.1 Global Design Issues

The most important characteristic of a data structure design is that it is *integrated.* Few decisions may be made with no reference to other parts of a design. To prevent paralysis, this section will discuss some issues that are common to all designs.

Some of the interactions among rules are cyclic in nature. For instance, the memory needed by objects determines the size of the pointer, which may be a component of some structure, therefore it determines the size of the structure, possibly changing the amount of memory required by objects. Convergence of this process cannot be proven in general, but there is little cause for worry, since iterations are typically matching linear growth in requirements vs exponential growth in resources, and resources will outstrip requirements quickly.

4.1.1 Feasibility

The very first task the designer should undertake is to see if the capacity of the machine is sufficient for all the data it is to store. Bits are rather fine-grained, but provide a common basis for comparison:

Is the number of bits in the machine at least as great as the number of bits required by the datatypes?

The answer to this question depends on how bits are counted. The number of bits in an integer n is just $\log_2 n$ rounded up; sequences multiply, structures add sizes of components, and so forth. The size of the machine must be carefully calculated. For instance, the VAX theoretically offers a 32-bit address space of 8-bit bytes or 2^{35} bits total, but 3/4 of it is reserved for various purposes [40], so the true addressable space is only 2^{33} bits. Note that this rule also ensures that infinite datatypes will be disallowed. The counting will be inaccurate, but at this stage, only approximations are needed.

Also, any basic operations like arithmetic should have hardware counterparts in some form. The basic operations are usually mathematical, arising from the ADT `integers`. Ideally, a full designer should be able to derive bignum algorithms from limited precision arithmetic, but this is asking a lot, considering that the problem is not easy even for human designers!

Do all the primitive operations specified by ADTs have machine operations to implement them?

(In the current implementation, this test is implicit; a missing operation will cause the coding phase to fail to generate code, ultimately resulting in an undefined function.)

4.2 Machine Characteristics

How does the designer decide that a VAX is a 32-bit machine? Is a 68000 a 16-bit or 32-bit machine? Reading manufacturer literature will surely lead to drastic overestimates, so we will need heuristics to guess at what the right sizes might be.

"Registers" include whatever storage places are used by compiled code, for function protocol, temporary values, and so forth. This should exclude condition code bits. This information must be specified in some fashion, but it will not be very complicated. It will also not actually be part of the machine or type description, but derives from the compiler algorithms, interface protocols, and so forth.

"Memory," however, can generally be found by a single heuristic that excludes large register banks and tertiary memories:

The "main memory" of a machine is a vector of at least 500 numbers each at least 6 bits in size, and which has at least one instruction that can dynamically access and modify the elements of the vector in constant time.

The size requirement effectively excludes registers, while the accessing requirement ensures that some sort of pointer addressing is possible. This would be necessary to exclude unusual machine structures, such as a serial buffer that might be large, but unusable as a heap.

Many subsequent rules mention "natural sizes" of the machine. Most modern hardware offers several sizes of objects to operate, such as 8-bit, 16-bit, and 32-bit operands. Given this, it does not always make sense to specify a single "word size":

> *"Natural sizes" include the sizes of registers, of memory elements, and any size mentioned in more than one instruction. Collect these sizes into a list and use whenever a "natural size" is called for.*

Machines have more subtle properties, but these are too complicated to describe here, and are not used regularly by implementors anyway. Such properties include memory shared by different processors and performance differences due to caches and pipelining.

4.2.1 Sizes of Objects

Ideally, the size of objects/pointers should match the "word size" of the machine, so as to optimize speed and to avoid wasting space. There is one rule that absolutely must be satisfied:

> *The size of a pointer must be sufficient to address the specified maximum number of data objects.*

Another heuristic suffices to make sure the pointers are not larger than absolutely necessary:

> *The size of an object should be equal to or less than the size of the registers.*

This rule will not work well if there are different register sets of widely varying sizes, as for instance on the Cray-1, which has both 24-bit and 64-bit registers. This is a point of interaction with full language semantics, since the choice of registers depends on how the language system uses them (for function arguments, temporaries, and so forth).

In the following rules, the term "word" will refer to a unit of storage sufficient to contain one object. A more sophisticated approach would allow several sizes for objects, but this is far more powerful than usually necessary.

4.2.2 Object Tables

Under some circumstances, it may be worthwhile to let all object references go indirectly through some sort of table. This is nearly universal for symbols in Lisp systems, in order to provide a dynamic redefinition capability. Another use for object tables is found in Smalltalk-80, where the 16-bit pointers of the Alto cannot directly address one megabyte of memory.

The range of applicability for object tables is rather narrow, as indicated by the nature of the preconditions:

If size of the pointer object is insufficient to address all of needed memory directly, but is sufficient to point to the required number of objects, then have the pointer point to a table of addresses for the objects proper.

If all types of objects share some field or slot, then the object table may be a good place to put them.

If all objects indirecting through the object table share some component, then pack that component into the object table entry.

One reason is that the object table is very regular, while objects may vary in size, so the meta-rule about simplicity suggests that more regular things are simpler (and therefore faster) to work with.

The use of symbol tables to allow dynamic redefinition of functions goes beyond the scope of this dissertation, since it involves executable code and language semantics.

4.3 Design for Integers

Ranges of integers are the simplest form of number to implement, but they are not totally straightforward. The main problems arise from the representation of larger integers.

Small integers (fixnums) should always be represented directly, since they tend to be quite common, even in the "more symbolic" programs.

If the range of integers fits within the available part of a single word, then represent directly, using 2s complement if the hardware does.

"Available part" means any part of a word not already allocated for some other purpose (tags, mark bits).

Anything larger than the data field of a word is going to need a more general representation involving storage allocation. The most general rule implements variable-length integers (bignums):

If the range of integers is too large for fixnums, use a varying-length vector of small integers.

This rule is an example of "type-to-type transformation." The rest of the designing will be done by the rules concerning vectors, which can represent the length in different ways, pack elements, or represent the vector in turn as a list.

The algorithms for bignum computation are theoretically simple (see Knuth [87]), but the details of access to bigits ("bignum digits," the word-sized groups of bits making up the bignum), propagating carries, and so forth will be complicated in efficient implementations. White has an excellent discussion in [166]. Among the unresolved issues is the choice between sign-magnitude and 2s-complement representation within a bignum (see [151]). Although the fixnum rule would seem to apply here (i.e. match the hardware representation for bigits), propagation of sign bits is potentially more expensive than converting signed magnitudes to 2s-complement, so the tradeoff should be evaluated experimentally [J.L. White, personal communication]. In addition, bigit size is a difficult question:

Make bigits be one bit less than a halfword in size, or two bits less than a full word.

Full bignum representation is expensive, both because each bignum requires space for length data, and because the computation algorithms are all iteration-based and will do unnecessary steps for very short bignums. A useful compromise is the adoption of a fixed-length allocated representation, usually one word:

If the range of numbers fits within one machine word, use a one-word block of memory for the number.

A more general rule could use distribution information to decide some plausible cutoff for fixed-length vs variable-length representation, but I do not know of any such situations that apply beyond individual programs.

Mindless application of these rules (as exemplified by the typical "AI" program) will likely result in a language system having *only* bignum representation—even for numbers like 0 and 1. The right approach requires more rules to set up automatically changing representations:

If a range of integers is very large (> machine word), substitute a sum of disjoint ranges, one small (< machine word) and two large, one containing all the larger numbers, the other containing the smaller numbers, and design each of these, defining arithmetic operations to dispatch and convert appropriately among the new ranges.

Historically, the division of positive and negative bignums into separate types is uncommon. At least one reason is the difficulty of writing the bignum code to handle two distinct types without making duplicates of the code. It has been more common to overlap integer ranges:

If a range of integers is very large (> word), substitute a sum of a smaller range (< word) along with the larger range, and define the operations to dispatch and convert types as necessary.

Defining the new operations is a critical step in both rules. The second rule introduces the possibility for the same integer to have two very different forms, so the operations not only need to dispatch on the type of integer, but also handle overflow of the smaller range and conversion from the bigger representation to the smaller one.

The schema for a range of integers actually allows arbitrary upper and lower bounds. In practice, the human implementor will pick the bounds to be $[-2^n, 2^n-1]$ for some n, which matches a 2s-complement representation, and everything works out fine. Occasionally, a different range will come up. There are two possibilities:

If the bits are available, expand an unusual range to 2s-complement bounds and represent using an appropriate rule.

If space is at a premium, then compression is the order of the day, even though operations will all be slowed down:

If there are only enough bits to represent the difference between lower and upper bounds of a range, and the difference is less than half of the absolute value of the range's bounds, represent the difference only and adjust all operations to subtract and add the lower bound before and after the operation itself.

These rules are especially useful in the design of integer representations to be used in other data structures; for instance, length fields in vectors or reference counts of objects.

4.4 Design for Sums

In a dynamic polymorphic language, type discrimination is a serious problem. Modern languages and compilers go to great lengths to eliminate any uncertainty about the type of an object. Common Lisp provides an extensive set of declarations for types, although compilers are not consistent in their use of them. Type inference algorithms have been extensively researched [28], although they can fail to disambiguate types completely, especially in the presence of complicated data flow, higher-order functions, and some forms of abstraction. In the end, some type discrimination will be needed.

There are at least four major ways to distinguish types dynamically:

1. Tagged pointers. Type is encoded in a bit field in the reference to an object.

2. Tagged objects. Type is encoded in a bit field in the object itself.

3. Separate spaces. Type is implicit in the address of the object.

4. BBOP ("Big Bag of Pages"). Contiguous small areas of memory are mapped into type codes, via a table.

Hybrid of these major approaches are quite common. It remains an open question as to which (if any) generally superior; BBOP seems to have the best characteristics overall. To date, however, tagged pointers are the most popular technique over all higher-level languages.

4.4.1 Design of Tags

Type tags inflict overhead on every object in memory. Therefore bit counting is very important. Although 1 bit out of a 32-bit word is only a 3% overhead in space, the more typical 5-bit tag consumes 15% of memory, which adds up to many RAM chips. In any case, the tag must be large enough to discriminate all the types:

The size of the tag field must be at least the base-2 logarithm of the number of types in the sum.

There might be good reasons to make it larger, though:

If the machine has a natural subobject size, and the data field remains large enough, make the tag be the size of the subobject.

This is for those cases in which the usable address space may be considerably less than the word width, as for instance on the original 68000, which supports 32-bit objects but has only a 24-bit address space. 8-bit tags are quite advantageous, since byte operations can be used to manipulate them. We should not forget language-specific hardware:

If a machine has special instructions to operate and dispatch on the part of a word exceeding the address space, make that part the tag field.

Another way to exploit machine characteristics is to know about about automatic high-order bit stripping:

If the actual address space of the machine is smaller than the apparent address space, make the tag field be the size of that difference.

A more unusual scheme assigns individual bits to each type:

If enough bits are available, then make each type correspond to a single bit in the tag field.

This may seem silly, but did find use in MacLISP, in the BBOP page table (so not in individual objects).

In theory, the tag bits could be scattered throughout a word, but the circumstances are special and will be handled in another way. So we can insist on using contiguous fields. The tag field may be allocated in either the high-order or low-order bits of a word, or in the middle. The middle has no advantages (I know of only one instance in which an implementor even considered doing this, in order to exploit a byte-swapping instruction), but the high and low ends of a word involve a large number of imponderables, and the final decision must be left to experimental evaluation.

The tag must be a contiguous field of bits, assigned to either the high-order or low-order end of an object/word.

(We will see shortly how to generate designs where both ends are used.) There is at least one obvious special case:

If the smallest addressing units of memory are smaller than object width by a power of 2, and the required tag size is less than or equal to that difference, then allocate that many bits in the low end of the object for a tag.

In most cases, this rule is advantageous for space only, since the bits must still be masked out to prevent address alignment errors. However, some hardware (such as the SPARC[1] [154]) can ignore those bits, thus rendering the tag removal unnecessary.

The assignment of tags to values can almost always be done totally randomly. There are a few instances where particular decisions can be beneficial, particularly in connection with low tags:

If low tags are in use and distinguishing less than four types, try all permutations of type assignments.

Actually, 8 types is more useful, but even 8! = 40320 designs is too many to evaluate. There is one tag value that stands out as especially important to assign properly:

Assign the tag value 0 to the most frequently-occurring type.

This rule is useful for both high and low tags, although it is slightly less advantageous for low tags.

If an integer type occurs frequently, and it includes both positive and negative numbers, and the tag is high-end, and there is an unassigned tag value, then subdivide that type and assign consecutive tags to positive and negative subtypes.

The last two heuristics together yield a very common design, in which positive integers are tagged with 0 and negative integers with -1 (all ones, considered as 2s-complement).

If the sum has sub-sums, then assign the tags topologically sorted order, assuming the root of the sum tree as the maximal element.

This trick ensures that many tests on groups of types can be implemented as range checks, thereby reducing the number of tests. For example, Common Lisp has a very elaborate hierarchy, with 10 or more types of numbers that must all answer "true" to the function `numberp`. A range check on properly assigned tags would require 2 comparisons instead of 10 equality tests. This has been exploited in Cambridge Lisp and other systems.

The "low tag as offset" pun can be handled directly here:

Adjust low-tagged pointers to be correct when the tag is present.

Storing the tag with the object, although a major technique in earlier times, only works if all objects have storage allocated to them. It does have the advantage of leaving the pointer "clean", so tag stripping and adding are not necessary.

If all types in the sum require some storage allocation, allocate a tag field in the memory block holding an object.

This rule will be more useful for specialized discrimination, perhaps among types of arrays or code objects.

---

[1]SPARC is a trademark of Sun Microsystems.

4.4.2 Design of Separate Spaces

Separate spaces are potentially more efficient than tags. They can be better for speed, because both pointers and immediate data can be used directly, and for space, since the type information is implicit in the address, and imposes no space overhead (unless large empty spaces are a problem). There is an advantage for the GC as well, since it can scan some areas very quickly, if they are already known not to contain pointers (such as strings and code blocks). Spaces fall down on type discrimination, since two address comparisons are needed.

Designing a separate space implementation is also quite simple:

> *Assign to each member of the sum an area big enough to hold as many objects as have been specified.*

The catch is that the sum of the allocated areas may exceed available memory, especially if the system is intended for use on large problems.

Address comparison is most expensive when the two addresses are completely arbitrary, and may have to be loaded from memory each time. One way to cheapen things is to assign on a 2^n boundary:

> *Round each assigned area up to some multiple of a power of 2, such that the varying part of the address is a "natural size".*

Halfword address comparisons are often faster, but this is usually too extreme of a solution, since the resulting spaces will be quite small. The best values are likely to be submultiples of a word, though this depends more on the instruction set than on the architecture.

Spice Lisp introduced the idea of "space-tag equivalence," in which the high address bits also look like a tag, thus giving the best of both worlds:

> *If the allowable address space is exactly the size of an object, then subdivide memory into as many areas as the next higher power of two over the number of types in the union, and define the predicates to work by tag checking.*

In theory, a pair of address comparisons in this scheme could be transformed into a single tag mask and comparison, but the reasoning behind this is rather subtle.

Finally, there is a small modification to immediate types that have spaces which are not actually used.

> *If a type is a range of integers with an immediate representation, then allocate the space consisting of addresses matching those integers to that type, or else modify the range's operations to add/subtract deltas to the space actually allocated.*

4.4.3 Design of BBOP

BBOP solves the initial allocation problem of separate spaces by breaking memory into a large number of small areas or pages, each aligned on 2^n boundaries and containing objects only of a single type. This retains the advantages of separate spaces while maintaining flexibility in the numbers of objects that can be allocated.

To distinguish types, divide memory into small equal-sized pages, make a page table equal in size to the number of pages, and make each page table entry large enough to discriminate all types but also be a "natural size".

The page table is additional space overhead, but not much; usually well under 1% of available memory.

A special issue for BBOP is the size of each page. The tradeoff is particularly clear, but not easy to decide. There is one rule that must be satisfied in any case:

The number of pages must be greater than the number of types.

To gain the most flexibility, there should be as many pages as possible, but to keep the size of the page table down, each page should be as large as possible. The product of these two is a constant—the size of memory, so optimization amounts to choosing a point on the hyperbola $xy = m$, where x is the page size, y is the number of pages, and m is the size of memory. Unfortunately, there is really no objective function to evaluate, so the actual choices are governed by other considerations. Intuitively, the best balance seems to be at $x = y$ (see Figure 4.1):

Set the page size to be the square root of the address space size.

Another way to look at this is to observe that the number of bits to address the page table is exactly a half-address, which is usually a "natural" size.

Machines with VM generally have small pages that are treated specially by the hardware, and there might be advantages to making BBOP pages the same size:

If the machine has paged virtual memory, make the page size be the same as the virtual memory page size.

This heuristic is more empirical than some of the others, since virtual memory behavior can be complicated and unexpected. The main purpose of the heuristic to reduce working set size due to BBOP pages extending over several VM pages.

Another force increasing the page size is the prospect of having objects so large that they must occupy several pages at once, which complicates everything:

Set the page size to hold the largest single object possible.

On the other hand, extremely large pages are effectively like separate spaces; there should be some variability in allocation:

The number of pages should be at least 10 times the number of types.

It would be silly to have so many pages that more of memory was dedicated to the page table:

The page table should be less than 10% of memory.

The 10% number is "soft," since it should be compared with the other typing techniques (if tags need 20% of memory, then a page table needing 15% of memory is still advantageous).

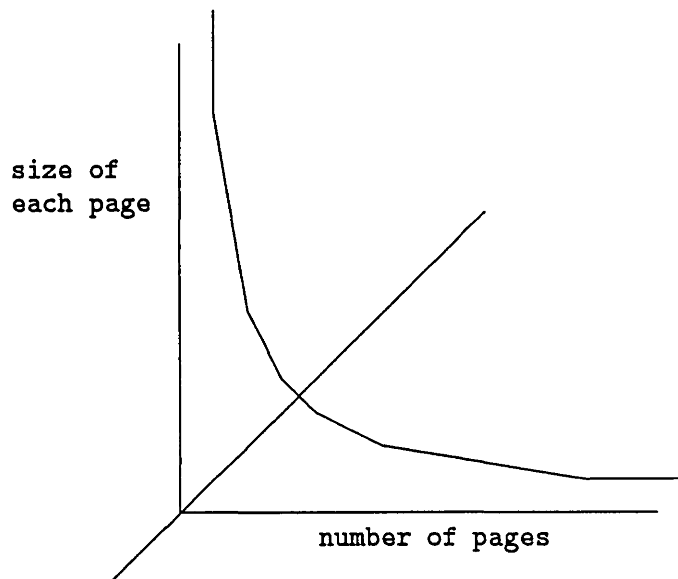

Figure 4.1. Graph of BBOP Tradeoffs

4.4.4 Combinations of Sums

Before going into some special techniques, it should be pointed out that the mindless application of these rules will cause problems if the language specification includes nested sums. The rules will have no problem choosing spaces for the first sum, tags for a subsum, and BBOP for a subsubsum. This is not always a ridiculous thing to do; many of the implementations in Chapter 2 have dedicated spaces for some types, and use tagging only in a heap space. Still, only certain combinations will work, as shown in Figure 4.2. The restrictions are recursive, so for instance once BBOP is used, all subsubtype discrimination must be done with tags, and once tags are used, they must always be used thereafter. (In practice, only one or two of these combinations will appear; simplicity is a virtue in runtime systems.)

These restrictions do not appear as separate rules; instead, they are extra unstated preconditions on all of the rules relating to type discrimination.

4.4.5 Merging and Splitting Sums

The preceding rules, if blindly applied to a sum of sums, will set out individual tags or spaces for each type in the sum, and additional tag fields or spaces for each of those types that is itself a sum. This is usually an unintelligent thing to do.

Consider the case of types

```
(defadt sexp (sum (numberp number)
                  (symbolp symbol)
```

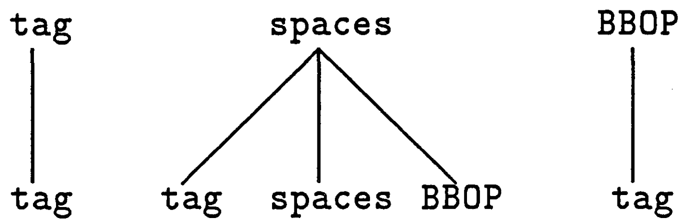

Figure 4.2. Allowable Combinations for Type Discrimination

```
                  (consp cons)))

(defadt number (sum (integerp integer)
                    (ratiop ratio)
                    (floatp float)))
```

Now suppose that the tag rules are being applied. First the sum **sexp** will be designed, and will use a 2-bit tag field to distinguish 3 types. Then, the tag rules can be applied again, again to set up a 2-bit tag field to distinguish types of numbers. However, this is a different tag field, so the total usage is 4 bits per word.

An alternative is to transform the source types into a single sum, something like

```
(defadt sexp (sum (numberp number)
                  (symbolp symbol)
                  (consp cons)
                  (integerp integer)
                  (ratiop ratio)
                  (floatp float))
  (define numberp (x) (or (integerp x) (ratiop x) (floatp x))))
```

Note that **numberp** is no longer primitive, but must be defined in terms of other predicates. In exchange for this, the tagged design will need only one 3-bit tag field, a savings of one bit overall. This may not seem like much, but the full Common Lisp type hierarchy is deep enough to result in 5 or more tag fields in each word, totalling up to perhaps 8-9 bits of tag field when 5 is sufficient. Thus the heuristic:

> *If a sum has a subtype which is also a sum, transform the two into a single sum.*

Repeated application of the rule yields a variety of new patterns of sums and subsums, up to and including a completely flat collection of types.

The inverse of this transformation may also be validly applied:

Divide a sum of more than two subtypes into two sums, one containing the other.

The effect, if tagging is used, is to make several smaller tag fields. This is very useful to get a sort of Huffman coding on type tags, particularly if some type just barely fits within a word. In practice, this technique commonly gives integers a shorter tag, thus allowing a larger range of integers to be represented within the same fixed-size word.

The ultimate reason for using multiple representations is improved time or space characteristics. Basically, savings can be achieved iff

$$C_0U_0 > C_1U_1 + C_2U_2 + DU_0$$

where C_i and U_i are the costs and usage frequencies of the initial representation 0 and the subrepresentations 1 and 2, (so $U_0 = U_1 + U_2$), and D is the cost for an operation to decide between the two. N-way representations can be modelled as $N-1$ two-way representations. The "costs" here are generic, and can be substituted with either time or space numbers, whichever is considered more important. (Note that D may have a significant cost in space as well as time, if the dispatching code is opencoded in many places.)

A more specific reason to split types is for the purpose of using low tag fields on byte-addressable machines. If the machine uses 32-bit aligned pointers, then the two lowest bits are unused, and make an excellent tag field. However, discrimination of 4 types is usually insufficient, and so additional type discrimination is needed. The rules can do this by splitting the main type into a sum of 3 types and another sum of all the remaining types.

Both splitting and merging are useful together, as a sort of general reorganization of type grouping. Also, different representation rules may be used on different parts of a sum. Typically, one or two types with special characteristics (symbols, compiled code) go into separate spaces, while the remainder are put into BBOP pages or are given tags.

4.5 Design for Structures

Structures are basically the same as C structs or Pascal records, and techniques for their implementation are well-known and relatively simple.

Assign components of a structure to random offsets in a freshly-allocated block of memory.

A less-common approach puts each component into a separate table, which cannot be done with just any typing scheme:

If the structure type is allocated into a separate space, then divide the space into tables, each of sufficient size to hold one component for all objects, and represent references to a structure as either a pointer or an index into the tables.

The pointer/index choice depends on the size of the table. An index is more compact, but more time-consuming to use, since it must always be added to the address of the table base.

Packing of small structure components is generally a good thing to do:

> *If two components are sufficiently small that together they are less than one word, combine them into a single word.*

This will accomplish the design of older Lisps on small-address machines, where 15-18 bit pointers to car and cdr were packed into a single 36-bit word. Recursive application of this rule will eventually try all possible packings of a multi-component structure, with exponential growth in the number of possibilities. Fortunately, few real-life structures have more than about five components, we would not expect to see torrents of minutely different designs arising from this rule.

Most machines have an "indirect" addressing mode memory directly pointed to, which is more efficient than the "displaced" mode necessary for addressing components in general. This can be exploited:

> *Assign the most frequently-accessed component to the 0 offset in the structure.*

Another way to represent structures is to think of them as vectors. This only works if each component of a structure can be forced into a uniform representation (either all slots the same type, or all slots subtypes of a single type, which will be the one used as vector element).

> *Represent a structure as a vector.*

This rule, along with a later one that represents vectors as lists, has the interesting consequence of using list representation for structures. This was once standard in Lisp systems, but since largely abandoned. In Prolog, records *vs* lists is an ongoing debate; see [27]. A somewhat less interesting result is an infinite regress, as a list element (a structure with two components) is represented as a vector, which is represented as a list, which is itself a structure of two components, *ad infinitum.* Again, mindless application of rules could easily cause this to happen.

Another indelicate rule splits a multi-component structure into two structures connected by pointers.

> *Transform a structure with more than two components into a structure one of whose components is another structure containing some of the components of the original.*

This could be useful if part of a structure were to be shared. Repeated application of this rule to one component at a time yields a list representation for a structure.

Conversely, structures with substructures can be merged into single larger structures:

> *If a structure has a component which is also a structure, transform the structure into a single structure with a merged list of components.*

If a structure component will never be changed after creation of the structure, then it can be handled somewhat more freely. Before going into details, we first need a rule that does not care:

Transform an immutable component into a mutable one.

This rule is more for bookkeeping purposes than anything else; its implementation creates a new name for the slot modifier that can be used during the coding process.

A structure may have any combination of mutable and immutable components, but this is awkward to deal with, so a convenience rule segregates the two:

Divide a structure into mutable and immutable substructures.

The immutable substructure can be implemented quite differently. The main advantage is that it need not be created anew, but a "new" structure could actually be a pointer to an old one. This is how interned symbols in Common Lisp are treated, and is also the idea behind the "hashing CONS" that has been used occasionally, where new cons cells are just pointers to old ones, if the car and cdr are the same as for some old cell. The idea of immutable objects is basic to functional programming, and the sharing of such objects is a key advantage of many evaluation schemes.

Alter the immutable structure's creator to try to reuse existing structures by hashing to them.

This rule immediately brings up the design of hashing algorithms to support this design. Like GC, the questions involved are complicated and space does not allow going into them.

4.6 Design of Vectors

Vectors here subsume objects of fixed and varying length, as well as those with mutable and immutable elements. We will assume vectors to be indexed from 0 on up; any adjustments for non-zero basing are simple and not of much interest.

As with structures, the simplest and most obvious representation is as a block of contiguous memory:

Represent a vector with a block of memory.

This rule fails to account for the length of the vector (perhaps the vector is of fixed length), and it says nothing about how the components are to be handled. The length must be recorded somehow for a varying-length vector, but it may also be necessary for fixed-length vectors as well, if references might possibly go out of bounds, or if something (such as the GC) has to be able to scan along memory. One way to record a vector's length is to record it explicitly:

Store the length as a field at the beginning of the vector.

This requires adjustment of the index, unless the pointer to the vector addresses the base of the vector's data:

> *If the machine has a capability for negative displacements, store the length below the address of the vector.*

The other thing that may be needed to be done to an index is to multiply or divide it. A great advantage may be derived from the use of low tags matching the size of words, then integer indices will already be shifted correctly. This will fall out of the coding phase, when some designs are much more efficient at vector accesses than others.

The other way to support varying lengths is to include an end-of-vector marker; an object of some type different from the components of the vector.

> *If there is some bit pattern of a size matching the vector elements, but is not itself a valid vector element, and either GC does not sweep memory or the vector elements are typed sufficiently for GC to identify them, allocate one extra position to each vector, and store that bit pattern there when creating the vector.*

Strings in C use a NUL (0) character, thus precluding the use of NUL within strings. In the case of Icon strings, this technique is not legitimate, since all possible characters may appear in a string. For general Lisp vectors, there may be a spare pattern not designating a normal object, such as a special "unbound" object. Another common solution is to use a new tag to designate a "non-object." Such an object or tag can also be used for unbound variables.

It is possible to use both end-of-vector and length fields at the same time. Modern Lisp systems that support interfaces to C can benefit from this, since passing a string to C requires nothing more than passing to the beginning of the string data, while getting the length is still a single access.

Packing of elements can and should occur in vectors. Strings are the most familiar example, but bit vectors are found in Common Lisp, and "word vectors" of various sorts prove to be useful in implementation of various internal structures (hash tables, I/O buffers, etc). Packing is simpler than for structures, since all elements are of the same type, but it is also more important, in that inefficiency exacts a heavier penalty. The tradeoff is again imponderable, since tighter packing may require more elaborate accessing code, but looser packing requires more space.

> *If the number of bits needed to represent a vector element is less than half the size of a word, then pack as many elements as possible into each word.*

The tightest packing may involve very uneven sizes with no actual space advantage, such as 7-bit characters in a 32-bit word. Going to a "natural size" loses nothing and gains speed:

> *If the number of bits to represent a vector element is smaller than a "natural size" that is a fraction of a word, then pack elements of the natural size into a word.*

This rule is essential to deriving a byte-packed form of Common Lisp strings, since characters are first-class data types, but using a full word for each character in a string is rather wasteful of space.

In special circumstances, allocation may not be necessary, such as for short bit vectors.

> *If the number of bits needed by a vector is small enough to fit in the available part of a word, allocate the word to hold the vector.*

Another rule concerns the size of the length field. A field sufficiently large to cover all sizes of vectors may be too large, compared to the average length of the vector. An example is symbols in Common Lisp systems, where nearly all symbols have names less than about 30 characters long, and most have 5–6 character names. The use of 1-byte length fields instead of 4-byte fields could save up to perhaps 50K bytes of space (which might or might not be significant). A disadvantage would be that two types of strings would then be required, since Common Lisp requires implementations to support strings up to 1024 characters in length, and recommends even higher limits. The rule just says to split types on the basis of length:

> *If most vectors are short, and if the maximum length is large, then transform the type into a sum of short vectors and full-length vectors.*

The basic representation of vectors is a bone of contention for declarative languages, since memory block representations has deep connections with von Neumann architecture. Both the functional programming and logic programming community have argued that representing vectors (and arrays) as blocks of memory is a bad idea. For instance, Wise [169] has proposed quadtree representations for matrices. List representation is valid too, at least for shorter vectors:

> *Represent a vector with a list, with each list cell containing one or more vector elements.*

Naively, one might expect that the list elements should be in the same order as in the vector, but there is no fundamental reason for this. There are justifications for reversing the list, or using a tree representation. For example, if the first elements of the vector can be shared with others (as in 3-Lisp's rails [144]), or if the later elements are accessed more often:

> *Represent a vector with a list in reverse order.*

Accessing random elements can be sped up by using a tree-structured representation, but this form also requires additional space:

> *Represent a vector with a tree-structured list.*

(A more powerful rule would split vectors into arbitrary sections linked together, but this does not seem to have any actual use.)

As with structures, immutability of elements has advantages. The advantages are potentially much greater here, since vectors may be quite large, and sharing a distinct space advantage. Icon and Snobol have always shared string contents whenever possible, so for instance taking successive substrings of a string does not actually cause any copying of string characters.

> *If separate spaces are in use, represent an immutable vector by making a space dedicated to the vector elements, and using a structure that contains a pointer to the beginning and a length, or else pointers to beginning and end of the elements.*

The real value of this rule derives from higher-level operations like concatenation and subvector extraction, which would be nonprimitives here. Evaluation needs to account for the performance of nonprimitives as well as primitives, but optimization from the abstract form to concrete code is again too hard for reasoning systems.

4.6.1 Vector/Structure Integration

Frequently a vector-like object will also have some structure-like slots. For instance, a Common Lisp vector will have `fill-pointer` and `adjustable-p` attributes, while Smalltalk defines its array objects to have pointers to their class. The implementor, however, must write the definition as a structure, one of whose components is a vector ADT.

The default behavior of the rules is to make a structure representation one of whose slots is a pointer to the vector representation, which is not bad, but does involve an extra pointer. Another approach is to use a single memory block with a fixed-length section at the front, followed by the varying part.

> *If the type is a structure one of whose components is a vector, allocate a single block of memory with the vector at the end of the block. Include a length field somewhere within the combined object.*

4.6.2 Arrays

As with sums of sums, vectors of vectors (or arrays) can benefit from special treatment. The original ideas about representation of arrays were developed very early on, but the first survey did not appear until 1962 [68]. The default behavior of the rules will result in recursive vector of vector representations, which may be actually be advantageous if multiplication is slow and sufficient memory for the extra pointers is available. I should point out that this representation may be more generally applicable than normally believed, since the extra memory overhead is significant only with certain shapes of arrays (consider that for a 1000x1000 array, the extra space is 1000 pointers or .1% extra). In addition, since the vectors can be moved around, large allocations are less frequent.

Indexing is the traditional method for array access, even though its main advantage is in space savings and possibly reduction of memory references:

If a vector has vectors as its elements, to some level of recursion, then represent as a block of memory addressed by multiplying and adding indices, preceded by a descriptor containing both lengths.

This rule covers situations in which, say, the first two indices index a block of memory each of whose elements is a pointer to a vector. Because it designs a representation rather than transforming types, it cannot be applied repeatedly to get higher-dimensional arrays.

The type-to-type transformation of structures into vectors means that a multidimensional array of structures has a special representation, where the component of the structue is just another index.

A somewhat dubious optimization allocates indexed array space along power-of-2 boundaries in order to change multiplication into shifting. Such a trick is rarely useful, since the space wastage is immense, compared to what may only be a minor speedup. Nevertheless, machines which multiply in software will benefit. It also makes sense when the array is expected to grow, since some growing room is pre-allocated.

If sufficient extra space is available, and arrays are not overlaid, then round sizes up to powers of two and use shifting instead of multiplication for indexed addressing.

A. Rosenberg has done some work of a fairly theoretical nature on the representation of dynamically changing arrays [129]. Some of the tradeoffs may seem highly unusual—some of the proposed designs sacrifice 99% of memory space in order to avoid copying array elements around. This may be worthwhile in APL, since many programs could potentially copy a single array hundreds of times while processing it. Unfortunately, this work was never tried experimentally [A. Rosenberg, personal communication].

Sparse arrays are also well-known, but start to tread into the realm of application program design, since their desirability is strongly dependent on the kinds of programs using such arrays.

4.7 Other Types

A number of special combinations arise that are not directly associated with particular schemas.

4.7.1 Floating Point Numbers

The subject of floating point representation has been intensively studied, but is not so important to us, since the decisions are usually embedded in hardware, and language implementors primarily concern themselves with efficient interfaces to that hardware. The IEEE standard [118] specifies both the behavior and the representation of floats, leaving very little to the imagination. Thus the first rule:

Figure 4.3. IEEE Floating Point Format

If the system has a builtin representation for floats, and the size of the float is compatible with the specified type, then represent floats that way.

This rule could be easily justified by comparing the speed of computing with the hardware representation compared to sequences of instructions.

In the formalisms of this dissertation, floats are actually small structures with integer components. As such, the structure rules will take over, and be totally unaware that the behavior may already be available. On the other hand, Common Lisp allows a variety of floating point formats, from short floats (with potentially as few as 13 bits of precision and 5 bits of exponent [146, p. 17]), to long floats of potentially unbounded precision (S-1 Lisp has up to 144-bit floating point numbers [24]. It is unlikely that hardware will be able to support exactly the types defined by the language, particularly in the case of short floats. If floats are smaller than the builtin size, then represent as ordinary structures and alter the operations to convert to the builtin size and back. If the floats are larger, then full software support will be necessary, again something beyond the simple rules here.

An alternate representation of floats is the slash representation [104], in which numbers are fractions in a fixed range. This representation has both advantages and disadvantages, but the evaluation criteria take us into serious numerical mathematics and far from the concerns of this work.

An unusual possibility concerns the problem of floats being too large for immediate representation if tags are also used. The IEEE standard makes a large number of 32-bit patterns undefined (Not a Numbers, or NaNs), exactly those for which the exponent is 255 and the 23-bit significand is non-zero, as illustrated in Figure 4.3. Some patterns are reserved to flag exceptions, but the number of exceptions is rather small, leaving some 8 million bit patterns unused. If a system did not require large numbers of data objects, the IEEE NaNs could be used to represent all other types, say with 3 bits for a tag, which leaves enough to address a million bytes or words. The advantage would be immediate representations for all floats while retaining all significant bits. This would work even better for double precision floats, since the number of spare bit patterns is larger.

Unfortunately, the standard permits NaNs to be used in rather arbitrary ways by floating point hardware and systems software, for instance using different bits for each kind of exception, so prospects for this form of representations may not be particularly promising.

4.7.2 Rational Numbers

Ratios of integers are not complex to represent (just pairs of integers where the denominator is never 0), but there is a question relating to converting ratios into lowest terms. Taking Common Lisp as an example, functions that get the numerator and denominator must return the lowest terms, but no other part of the specification requires the ratio to be *internally* represented in lowest terms. In fact, converting to lowest terms after each operation is expensive, and it might be better to wait until the numerator and denominator were about to become bignums, and only then reduce to lowest terms.

4.8 Special Considerations

The topics here do not fit well with the rules covering particular schemas.

4.8.1 List Compaction

Lists are not directly expressible using one of the type schemas that have been defined. However, the characteristics of entire lists may be different from those of list cells considered individually. It has long been known that the cdr of a list cell is more likely to contain another list cell than an atom [22]. The question of what to do about this remains open—although hardware has been designed to optimize list representations, studies do not agree on its overall effectiveness.

The most common technique to exploit this regularity is known as cdr-coding. It involves the definition of an alternate form of cons cell, in which the car is of normal size, and the cdr is only a few bits. The cdr is a relative address, pointing to the next word or possibly the word after that. There may also be a bit pattern for the NIL at the end of the list. Cdr-coding has some obvious problems, such as potential inefficiency in the face of destructive operations.

Another approach to compact list storage is to use 3-element list cells. P. Sipala has done a very sophisticated analysis of this alternative [143], and concluded that 12% storage savings could be gained, assuming the average numbers reported in Clark's measurements, which is comparable to the savings from cdr-coding. Historically, this seems to have been done only once, in UT LISP. 4-element list cells have also been used by Takeuchi and Okuno [155]. In the formalism of this dissertation, however, the possibility for multi-element list cells could only arise by partitioning vectors; there is no way to preserve normal list semantics with 3-element cells.

To generalize this issue, we can consider any sort of coherent behavior among large sets of objects of the same type. Opportunities include symbol spaces (certain slot values showing up frequently).

4.8.2 Storage Reclamation

Garbage collection (or more generally storage reclamation) is a general concern when memory is "used up" in a dynamic fashion. All of the languages considered

in this dissertation have automatic allocation and deallocation, and do not provide for explicit manual deallocation (see [6, p. 283] for a remark on this).

Reclamation may not be necessary. This can happen if the allocation happens at a rate that will not exhaust available memory before the program completes execution. In practice, this may happen in two ways: either allocation is rare or memory is very large.

Allocation tends to be frequent in higher-level languages, but sometimes is reduced when doing extreme optimizations (such as for Lisp-based editors). "Static" allocation may be observed in Lisp programs when large data structures like arrays are allocated at the start of execution, then destructively modified thereafter. Destructive operations in general can reduce the allocation rate to as low as a few list cells per second (at the price of program clarity). Such economy requires considerable programming effort, and usually has the effect of transforming a "high-level" program into something resembling C code.

Another way to avoid reclamation is to use very large memories, an idea which seems to have been suggested first by White [165]. The idea is that in a large virtual memory (32-64 bits address space), the unreachable data structures gradually end up swapped out to disk. When the program is finished, memory can be discarded, including the swapping space. A variation would be to preserve the program at some convenient point (say between toploop commands), exit, then restart. This approach was actually used on the MIT and Symbolics Lisp machines for several years, because the supplied garbage collector was buggy [110]. Even so, large VMs have their own problems, such as fragmentation and working set size limitations. Any kind of coalescing process is going to have overheads comparable to GC, so is not a very good solution.

Despite the drawbacks, both approaches to eliminating reclamation should be considered by the designer, since even the fastest reclamation techniques still have a substantial effect on performance. When considering these issues, it is important to know the rate of garbage production, the size of available memory (real and virtual), and the expected running time of the program. The total expected amount of consumption (rate times execution) is a quick test—if larger than available memory, then reclamation will certainly be necessary.

> *If total memory usage by persistent objects over program execution might exceed the available memory, then include a reclamation mechanism.*

(Stack-allocated objects need no special reclamation machinery.) Note that this rule might only be applied to particular parts of storage. For instance, list cell reclamation is a given in Lisp, but compiled code reclamation is relatively uncommon—many systems will just fail if the compiled code space is filled up.

If reclamation is required, there are a variety of ways to implement it. These can be classified by when and how much storage is made available for re-use. On the one hand, storage can be made available the moment it is released (as in reference counting), or all of it can be made available when memory is about to be completely used up (as in garbage collection). Mixed algorithms are also possible.

An independent dimension of reclamation is its *leakiness.* This refers to how good an algorithm is at recovering unused space. A leaky reclamation method is not necessarily a mistake—if the rate of leakage is low enough, it may never become an issue. Leakiness is really a generalization of the reclaim/no reclaim decision. It is also possible to use several reclamation methods; a fast but leaky algorithm could be backed up by a slower but more efficient method. Backing up reference counting with a full GC is one well-known choice.

> *If a reclamation method leaks unacceptably, add a more leakproof reclamation method to be invoked when storage appears to be exhausted.*

The decision between reference counting and garbage collection is a difficult one, but although implementors seem to have exercised themselves over the question, the decisions rarely have any facts to back them up. Reference counting has the advantage of no interruptions in execution:

> *If there are no mutable/circular structures, use reference counting.*

Unfortunately, the "smoothness" of reference counting cannot be expressed in ADT schemas, not even with additional pragmas.

If reference counting is chosen, then the size of the count field must be decided:

> *If reference counting is to be leakproof, the size of the count field must be at least enough to count all objects in the system that might reference any given object.*

Deciding where the count is to be stored is a thorny problem. Fortunately, not all objects need a count field:

> *If an object includes or reference an allocated piece of memory, then include a count field with the object.*

An object table is a good place, as is the header word, if there is room.

Garbage collection is a complete area of research in itself [34,86]. Rather than be swamped in the complexities of unusual algorithms (which are being discovered all the time), we will only consider rules for familiar techniques. First, the reasons for doing GC:

> *If many circular structures are possible, or the average amount of sharing is high enough to require large count fields, then choose garbage collection.*

Garbage collection is based on the notion that all data in use can be traced from some central known place, and that unused space is whatever is not so traceable. Once the unused space has been identified, it must be made available for use again. The process may be done all at once, or part at a time. It is normally initiated by allocation attempts:

> *Attach the garbage collection invocation to allocation operations.*

(An alternative might be to have GC proceed at regular intervals, initiated by clock interrupts, but this is rather speculative.)

The tracing process derives directly from the type definitions, whether it uses a stack or pointer reversal. It must also know whether each type has an immediate or allocated representation.

Marking is somewhat more interesting. One way to view marking is as a "type discrimination" problem, where there are two types of objects: traced and untraced. With this point of view, we can apply the general rules for the representation of sums, and derive the use of a mark bit, either in the pointer or in the object. We can also hypothesize marking techniques based on separate-space-like behavior or BBOP-like behavior. Conversely, the popular technique of using a bit table for marking can be generalized to general type discrimination. With such a scheme, all the tags of all objects would be stored in a separate area of memory (as opposed to BBOP, which does some grouping as well).

Marking via separate spaces actually translates into the familiar technique of stop-and-copy, which is based on the use of two spaces, one for old objects and one for new copies. Objects in the old space get copied, while objects in the new space are left alone, and the decision is made by comparing addresses. As with regular separate spaces, sufficiency of address space is an issue:

> *If available virtual memory is more than twice the required memory and representation is tagged (maybe BBOP?), use stop-and-copy.*

The test for the extra space needs to be more delicate in the case of separate type spaces:

> *If available space is more than twice the required space for each type in spaces, use stop-and-copy.*

The idea of BBOP marking derives from stop-and-copy in the same way that BBOP for types is an improvement on separate spaces. We identify each page as "old" or "new", and copy each page individually. This technique does not seem to have been used, perhaps because the advantages of BBOP are less important for GC.

Generation scavenging is an elaboration based on the observation that many objects become garbage soon after creation. From our point of view, this amounts to creating several types of marks, and type discrimination techniques are still relevant. For instance, SPUR uses tags to distinguish generations.

With stop-and-copy, the old space need not have anything done to it. With mark bits, something must be done to find the unused space, usually a linear sweep through memory, which imposes a serious constraint on all object representations:

> *If sweeping in use, then design other objects to be identifiable on a linear pass in each GCed space.*

Assuming that unused objects can be identified, there is then the question of how to make them available again. The best technique is the use of some sort of free list, but this has only limited applicability:

If all objects of a uniform or nearly uniform size, or if fragmentation is not a problem, then chain unused objects together into a list that allocators can traverse to get new space.

Still, the highly uniform cons-based representations of early Lisp systems meant that GC was relatively fast. Free lists are still common in the storage managment of symbols and sometimes cons cells, if they are stored in a separate space. If free lists would cause fragmentation, then compaction is the thing to do:

If all of memory needed and varying-size objects are present, compact used data into one end of a block of memory, allocating a relocation table or a slot in each object.

As with marking techniques, the relocation address falls under all the structure rules, for its handling. An additional possbility is that a little-used structure slot might be borrowed for relocation. This was not uncommon on older machines with large words and small addresses, but at present, relocation tables are more likely to be separate.

4.9 Summary

The design rules here are not unusually complicated or subtle. Part of the reason is that the interactions among rules are more important than the rules themselves, which also means that most design alternatives cannot be compared until coding and evaluation. Not all of the rules have been tested in a designer, so they are likely to be missing important, particularly the rules relating to storage recovery techniques.

Some heuristics are so general that they do not appear as single rules, but as collections of rules with a common theme. The common themes could be thought of as meta-rules:

- Transform one type specification into another.
- Use the simplest possible representations.
- Exploit statistical patterns of usage.

These meta-rules cannot be implemented directly in anything less than a fully automated reasoning system on the scale of Eurisko [93], but they are quite useful in understanding the motivations for specific rules, as well as for suggesting new rules.

Type-to-type transformation proves to be an important mechanism for making decisions. This was not anticipated at the outset, and does not seem to have been considered seriously in previous ADT research. Certain decisions (such as the use of a single tag field) can only be rationally explained by transforming the type structure, while from a more practical viewpoint, many design rules are simplified by being able to make a small decision and then rely on some other rule to do the actual

construction of code. Informal application of type-to-type transformation can also be useful: the observation that garbage collection involves a type discrimination step allows us to "borrow" the type discrimination rules and use them in designing garbage collection techniques.

Simplicity of representation is a common theme, though perhaps more notable by the absence, rather than presence, of particular rules. Since the addition of even a single machine instruction to a primitive can measurably affect overall program performance, complexity *must* be avoided. Some puns are motivated by this meta-rule, the rationale being that if something has two meanings, then special code is not necessary to distinguish one meaning from the other. On the whole, the meta-rule eliminates interesting but overly complicated representations and the rules that might generate them.

Extra complexity may be worthwhile if general but poor representations are needed in only a small fraction of operations. Huffman coding is the information-theoretic version of this meta-rule. Since extra complexity also incurs new costs while reducing others, there is an important tradeoff for each such case.

These rules will not and cannot make the final decisions about data representations. The many apparent contradictions underscore this fact: each pair of contradictory rules give rise to multiple plausible designs. The implementor can only make a rational choice by evaluating each design, either by hand simulation or by testing in a real system. This will be the subject of the next chapter.

CHAPTER 5

AN AUTOMATIC DESIGNER

A host of ... compilers ... darkened the face of learning.

E. Gibbon, *Decline and Fall of the Roman Empire* (1783)

While the heuristic rules of the previous chapter can provide much guidance to the human designer, the real test of their validity is incorporation into a program that can produce its own designs, and the usage of those designs in a real language system. In this chapter, I will concentrate on Common Lisp; although in principle, the designer could design for any language, the output is restricted to definitions of functions and variables. In most cases this will be satisfactory (arithmetic primitives in Prolog are fundamentally functional in nature, for example), but in general the form of the design will have to conform to the language semantics. The machine code segment of the designer is definitely restricted to Common Lisp, in fact the syntactic form of its output is useful for only a single implementation; the state of the art in language implementation does not allow for portable definitions of code generation patterns.

The experimental approach described here has three main parts, each supported by a separate piece of software:

1. Design. The designer program uses the previous chapter's heuristic rules to produce a number of designs, which are collections of function definitions in an abstract Lisp-like language.

2. Code. The coder program turns the abstract code of the designs into Common Lisp, or machine language or both, as appropriate.

3. Evaluate. The coder's output can be incorporated into a Common Lisp system that has been designed for this purpose.

The real implementation that will be used is a Common Lisp system that has been under construction at Utah since late 1986, dubbed Utah Common Lisp or UCL. It has been specially designed to be highly alterable, from the data-driven frontend to the portable micro-kernel. In particular, most of the compiler's code generation is done by declarative forms, which can be changed to implement a wide variety of representations. The forms of interest here are all short sequences

of machine language that implement single functions or variables. The compiler substitutes these bits of machine language for calls to primitive functions, in a process called *opencoding* (also known as "inline coding"). In the best case, an entire program may turn into machine code without a single function call. This is important, because setting up and performing a function call may take many instructions, which will overshadow the one or two instructions in the function body that actually do the desired computation.

Before delving into the detailed description of the code, the reader should keep in mind several basic decisions: 1) since the designer/coder is a one-time-only sort of program operating on small amounts of data, time and space efficiency are unimportant, 2) the designer is heuristic, meaning that it is not expected to handle the most unusual situations, and 3) things have been greatly simplified in order to get a working program. Later sections will discuss the limitations in more detail.

5.1 The Designer

The designer starts with the source and target ADTs. The source ADT will generally be something resembling Lisp datatypes, while the target ADT will be something resembling a machine. The designer will not produce incorrect results if given other types, but it would very likely fail to generate any designs, due to the lack of appropriate rules. Still, inverting source and target could yield data structures for a machine emulator in Lisp, or using another Lisp-like target could produce designs for embedded languages—see the discussion of rails in [144,138] for an example of a nontrivial representation decision in an embedded language.

The designer output is a list of *design objects* (or simply *designs*), each of which references a number of *primitive objects* (or *primitives*), which are the functions and variables that the design defines. Each design is self-contained, with the exception of ADTs, which are referenced by name and shared by all designs. Each primitive is referenced by exactly one design—although it is possible for several designs to share a primitive, this is unusual, since the definition of a primitive is almost always different from one design to another.

The design process consists of two phases. The first is decision making, in which the design decisions are made to create the designs, while the second phase builds the complete design code for primitives. When done, the designer writes out the design code into a file (for debugging purposes).

The designer is written in Lisp. An earlier version was written in MRS [132], using essentially a Prolog subset. The chief disadvantages of this approach were 1) the substantial amounts of computation required, which required either auxiliary functions or lengthy rules, and 2) the sheer quantity of the information generated. Each design is perhaps a page long when printed out; if done with a standard logic programming language, it must either be a large single term or a collection of predicates. In the first case, rules now have to repeatedly assemble and disassemble terms (albeit via unification), and in the the second, predicates have to be asserted into the database, which brings its own set of problems. The ideal language would be a sort of object-oriented Prolog, but these are not yet generally available.

5.1.1 Making Decisions

The basic design algorithm involves a set of *choices*, which are essentially rules that make design decisions. Each choice is self-contained; it is a function that takes an incomplete (*partial*) design as its argument and returns a (possibly empty) list of partial designs. The algorithm just exhaustively applies each choice to each partial design until all the designs left have at least one goal "done" and no goals "undone". Although this approach is not very efficient, the number of different choices is unlikely to be much over 50. The algorithm also cleans up the "done" and "undone" slots of a design by treating them as sets and removing duplicate elements. This keeps certain loops from occurring, by eliminating some goals that have already been achieved (for instance, when a recursive type is being designed). The algorithm applies itself recursively to the list of designs returned by a choice. Completed designs go onto a list `*designs*` and are not worked on further.

The problem with this obvious and exhaustive approach to design is that it can generate astronomical numbers of designs. For instance, random assignment of PSL's 19 types to 5 tag bits results in

$$32!/(32-19)! = 6645143393563388313600000$$

different assignments, each of which could be considered a different design. The performance of the different designs is identical in *almost* all cases. The exceptions are puns, and there are only a few (for tags, the assignment of zero is one important pun). Since I have not been able to figure out a general way to prune such gigantic design spaces into something more manageable, I have taken the alternate tack of making the design rules "know more," and so in the case of tags, assignment is cycled rather than permuted, thus assigning each tag to each type once but no more. In the case of PSL, this would result in 32 designs; still a lot, but manageable.

5.1.2 Choice Objects

Choices themselves do all the real work. They always have an if-then form, although this is not required. The typical test is for a type of schema, or perhaps combination of schemas, along with values for size/frequency pragmas. If the test succeeds, the choice usually *copies* the partial design and modifies the copy. Some choices will make several copies; in any case, any new partial designs come back in a list. An empty list indicates failure. The original design should not be returned. The copying is necessary to support the generation of multiple designs, which will only share some parts of their structure. (This is not as wasteful of space as it might seem, since only the design objects and primitives themselves are copied, not the contents of their slots.) The set of modifications to the copies is generally the same:

1. Record the making of the choice in a list in a dedicated slot. Exact format is unimportant, since the data is for documentation only.

```
(defchoice hi-tagged-sum (d)
  (let ((type (undone-type 'sum d))
        (newdesigns nil))
    (if type
      (let* ((n (length (cdr (adt-schema type))))
             (lis (iota n)))
        (dotimes (i n)
          (push (one-hi-tag-assignment
                 d
                 type
                 (append (nthcdr i lis)
                         (ldiff lis (nthcdr i lis))))
                newdesigns))))
    newdesigns))
```

Figure 5.1. Tag Assignment Choice

2. Dissect the schema(s) that matched the test, and create an object for each primitive mentioned in the schema(s).
3. Fill in each primitive's *basic code*, which is the operation with no adornments of any sort.
4. Create any *wrappers* and initialization code necessary. Wrappers usually handle some sort of generalized type conversion.
5. Record a goal as having been done and possibly add new goals.
6. Add the new partial design object to the list of those to be returned.

Figures 5.1 and 5.2 illustrate a complete choice object. This one designs high tags for sums. The function `undone-type` matches if there is a type not already designed, and which has a `sum` schema. If so, then the choice will generate several different tag assignment patterns, in which the values are cycled around so that each type gets a 0 tag in some assignment. The function `one-hi-tag-assignment` actually does the work of building the abstract design code, starting by making a copy of the design and its primitives, using `copy-all-design`. The only functions to be generated are predicates, one for each subtype, along with their signatures (indicated by `->`). Wrappers must be defined to convert objects from typed to untyped and vice versa (this will be explained further in the next section). Also, the use of tags implies a uniform heap, and the choice generates initialization code and variables for it.

5.1.3 Type-to-Type Transformation

Type-to-type transformations are a special case of choice objects, with simpler behavior. Basically, instead of creating parts of a design, a transformation alters

```
(defun one-hi-tag-assignment (d type assign)
  (let ((machine (design-target d))
        (newdesign (copy-all-design d)))
    (add-decision (cons (adt-name type) (cons 'tagged assign)) newdesign)
    (let* ((fns (mapcar #'car (cdr (adt-schema type))))
           (subtypes (mapcar #'cadr (cdr (adt-schema type))))
           (tagsize (ceiling (log (length subtypes) 2)))
           (datasize (- (machine-word-size machine) tagsize))
           (tagmask (ash (1- (expt 2 tagsize)) datasize))
           (datamask (1- (expt 2 datasize))))
      (do ((resti assign (cdr resti)) (i (car assign) (car resti))
           (restf fns (cdr restf)) (restt subtypes (cdr restt)))
          ((null restt))
          (add-predicate (car restf)
                         `(-> (,(adt-name (design-source d))) (boolean))
                         `(lambda (x)
                            (eq (logand x ,tagmask)
                                ,(ash i datasize)))
                         newdesign)
          (cond ((zerop i)
                 (push (cons `(-> ,(car restt) ,(adt-name type))
                             `(lambda (x) x))
                       (design-mappers newdesign))
                 (push (cons `(-> ,(adt-name type) ,(car restt))
                             `(lambda (x) x))
                       (design-mappers newdesign)))
                (t (push (cons
                          `(-> ,(car restt) ,(adt-name type))
                          `(lambda (x) (logior x ,(ash i datasize))))
                         (design-mappers newdesign))
                   (push (cons `(-> ,(adt-name type) ,(car restt))
                               `(lambda (x) (logand x ,datamask)))
                         (design-mappers newdesign)))))
      (push `(let ((space (allocate-proto-heap 10000)))
               (setq heap-lower-bound space)
               (setq heap-upper-bound (fix+ space 10000))
               (setq next-free-heap space))
            (design-init-code newdesign))
      (add-variable 'heap-lower-bound 'integer newdesign)
      (add-variable 'heap-upper-bound 'integer newdesign)
      (add-variable 'next-free-heap 'integer newdesign)
      (add-sub (adt-name type) subtypes newdesign)
      newdesign)))
```

Figure 5.2. Generator of Tag Assignment Code

the source type so as preserve semantics. The creation of a new type is encapsulated in a single function `copy-all-adt`; the Common Lisp function `gentemp` produces the new names needed. (Note that tangled type structures will need alteration of references as well as changes in the name slot.) The design has slots for both the original source type as well as for the current source type. Figure 5.3 illustrates a type-to-type transformation that merge sums of sums into a single sum type. The first part does matching, and is complicated by the need to search for a subtype that is itself a sum. The new schema is simple to construct, and the only other work needed is to define the predicate that is now no longer a primitive.

5.1.4 Finishing the Design

The basic code fragments for each primitive are not complete definitions. The basic definition of `car`, for instance, just says how to follow a pointer—there is no mention of tag stripping, type checking, or anything else. However, a complete primitive may be moderately complicated; in a tagged implementation, most operations start by removing the tag, then do an operation, and add tags back into any results. This is not part of coding, since the composition of these operations requires global information about the design, and is independent of any particular target machine.

It turns out that we need *two* notions of type for a function. The first is the conventional notion of type—`car` is defined on cons cells, but not on numbers. The second is something that I will call the *context type* (in the apparent absence of such a notion in the literature). The context type of a function says what types *might* have to be recognized or returned, irrespective of what types can actually be manipulated. It is a generalization of the notions of "raw" objects vs typed objects that are important to optimizing Lisp compilers. Normally, user-visible functions in Lisp systems have a context type that includes *all* possible objects (the type `t` in Common Lisp). In other words, any object may be passed to `+`, and it will do the operation or issue an error message; there is no possibility of, say, a pointer to a cons cell being mistaken for a number. In compiled code, however, `+` may appear in a context where the object being passed is already known to be a float, so its type info need not be checked/removed/added. For this use of `+`, the context type of its argument is `float`. In some other case, an argument may be known only to be a number, but not which kind, and the compiler might be able to eliminate some error checking.

Aside from speculative possibilities for compiler optimization, context types are important because they determine how to construct a complete function definition. If the context type of a code fragment does not match the context type of the primitive, then "wrappers" must be added to make them match up. Tag stripping/adding operations are perhaps the most common wrappers; they appear as shown in Figure 5.4. Manipulation is straightforward, since the wrappers are unary functions. It may be that several wrappers are necessary; in a tagged system with more than one tag field, there will be a separate wrapper for each field. If

```
(defchoice merge-subtypes (d)
  (let ((type (undone-type 'sum d)))
    (if type
      (let ((sch (adt-schema type)))
        (if (and (eq 'sum (car sch))
                 (some #'(lambda (sub)
                           (eq 'sum
                               (car (adt-schema
                                     (find-adt (cadr sub))))))
                       (cdr sch)))
          (merge-subtrees d))))))

(defun merge-subtrees (d)
  (let ((main (find-adt (car (design-undone d))))
        (results nil))
    (let ((sch (adt-schema main)))
      (dolist (sub (cdr sch) nil)
        (let ((ss (adt-schema (find-adt (cadr sub)))))
          (if (eq 'sum (car ss))
            (let ((newdesign (copy-all-design d))
                  (newtype (copy-all-adt main)))
              (setf (adt-schema newtype)
                    (cons 'sum (mapcan #'(lambda (s)
                                           (if (equal sub s)
                                             (cdr ss)
                                             (list s)))
                                       (cdr (adt-schema newtype)))))
              (push (make-primitive
                     :name (car sub)
                     :type `(-> (,(adt-name main)) (boolean))
                     :basic-code
                     `(lambda (x)
                        (or ,@(mapcar #'(lambda (s) `(,(car s) x))
                                      (cdr ss)))))
                    (adt-functions newtype))
              (add-sub (adt-name main) (list (adt-name newtype))
                       newdesign)
              (push newdesign results))))))
    results))
```

Figure 5.3. A Type-to-Type Transformation

```
[type]    svref is (-> (vector integer) (t))
[context] svref is (-> (  t        t    ) (t))

(define svref
  (lambda (x i) (lref (+ x (* 4 i)))))

   ||
   \/

(define svref
  (lambda (x i) (lref (+ (logand data x)
                         (* 4 (logand data i))))))
```

Figure 5.4. Addition of Wrappers to a Primitive

the context type matches the type coming from an operation, then no wrapper is necessary, as in the result of a `car` function.

Sometimes more powerful wrappers will be needed. For instance, a structure-creating function must pass a type to the allocator when using spaces/BBOP, but need only pass a size to a general heap allocator, if tags are used. However, the structure-designing rules do not necessarily know which scheme will be used, so their basic code will need modifications more extensive than available by wrapping.

Wrapping by analysis of context type is a rather expedient answer to a general problem of ADT implementation, but seems to capture an important aspect of code generation from designs. The process does make problems for the coder however, which must attempt to take multiple masking and other operations and turn them into single fast instructions.

5.2 The Coder

Coding is essentially a process of compilation, since the task is to transform abstract Lisp code into machine instructions. The design code is essentially Lisp, but adapted as necessary for succinct expression—a more functionally pure form is desirable, but can also get too complicated for matching processes to succeed.

Although this would appear to be an ideal task for a compiler generator of the sort that has been studied extensively, there are no "off-the-shelf" systems that can be used independently of a complete compiler-writing system, so I have been obliged to write my own generator. The algorithm is extremely simple; first, the code is flattened into a series of assignment statements (essentially the quads of standard compilers), then each assignment is matched against descriptions of machine instructions, and rewritten into an instruction if the match is successful. If the match is not successful, the coder assumes that a nonprimitive is present

and gives up; the code will eventually appear as a Lisp definition (which might or might not be valid, but the coder has no way of knowing). In general, the coder is free to break a function up into arbitrary combinations of normal functions and opencoded ones. Although this is not usual for compiler generators, the process is greatly simplified because the regular UCL compiler can be relied on to handle conditionals and other more complicated constructs.

The coder works with three classes of primitive: variables, functions, and predicates. Functions and predicates are similar, but predicates return a "control state" rather than values, and the UCL compiler generates code differently for predicates inside of conditionals (if a predicate must produce an actual `t` or `nil`, the compiler will effectively compile a `(if <pred> t nil)`). Primitive variables are rather simple to handle, since they do not have any internal structure, just a type.

5.2.1 Division and Flattening

The first step is to recursively scan through the definition. Most special forms and calls to normal functions can be handled by the regular compiler, so the flattener should divide a definition into smaller pieces connected by function calls (this has not been implemented yet).

If a function can be opencoded, it then goes into a flattener that converts nested function calls into a sequence of assignment forms with single calls producing the value to be assigned (essentially the "three-address code" of conventional compilers). Flattening is rather simple-minded and introduces large numbers of temporary registers, but the UCL register allocator will filter these out, so they are not a concern.

5.2.2 Instruction Matching

Finally, each statement in the flattened code is matched against machine instructions expressed as rewrite rules (axioms) in the machine ADT. The result of each rewriting is a short (1–2 member) sequence of assembly language instructions. Again, although a single statement may turn into several instructions, the register allocator can clear up many situations, while peephole optimization could improve others.

Failures to match here are serious, since any nonprimitives should have been detected already. The coder should leave the primitive undefined.

5.2.3 Generating Files

The final step is to produce files that can be loaded and compiled by UCL. There are two files; the cn files contain opencodings defined with UCL macros, while the pn files are definitions for nonprimitives along with the circular definitions for the primitives. Figure 5.5 shows part of an opencoding file (the details will become clearer as the UCL compiler is described), while Figure 5.6 illustrates the primitives.

Circular definitions of the form

```
;;;; Machine-generated opencodings

(defopen
 fix+
 (move long (arg 0) (temp * 5))
 (move long (immediate long 2147483647) (temp * 4))
 (move long (temp * 5) (temp * 3))
 (and long (temp * 4) (temp * 3))
 (move long (arg 1) (temp * 7))
 (move long (immediate long 2147483647) (temp * 6))
 (move long (temp * 7) (temp * 2))
 (and long (temp * 6) (temp * 2))
 (move long (temp * 3) (temp * 1))
 (add long (temp * 2) (temp * 1))
 (move long (immediate long 0) (temp * 0))
 (move long (temp * 1) (result 0))
 (or long (temp * 0) (result 0)))

(defopen
 cdr
 (move long (arg 0) (temp * 4))
 (move long (immediate long 2147483647) (temp * 3))
 (move long (temp * 4) (temp * 2))
 (and long (temp * 3) (temp * 2))
 (move long (immediate long 1) (temp * 1))
 (move long (temp * 2) (temp * 0))
 (add long (temp * 1) (temp * 0))
 (move long (indirect (temp * 0)) (result 0)))

(defsysvar next-free-heap)

(defsysvar heap-upper-bound)

(defsysvar heap-lower-bound)

...
```

Figure 5.5. Machine-Generated Opencodings

```
;;;; Machine-generated primitives

(defun init-purelisp ()
 (let ((space (allocate-proto-heap 10000)))
   (setq heap-lower-bound space)
   (setq heap-upper-bound (fix+ space 10000))
   (setq next-free-heap space)))

(defun fix+ (x y) (declare (inline fix+)) (fix+ x y))

(defun fix>= (x y) (declare (inline fix>=)) (if (fix>= x y) t nil))

(defun cdr (x) (declare (inline cdr)) (cdr x))

...
```

Figure 5.6. Machine-Generated Primitives

```
(defun foo (x y) (foo x y))
```

look silly, but they are surprisingly common in compiler-based Lisp systems. Their purpose is merely to supply a function protocol for the opencoding. The idea is that although the opencoding suffices to compile direct calls to the primitive, both indirect calls (via `apply` and `funcall`) and interpreted calls need a real function definition. The compiler opencodes the call to `foo` in the circular definition, while handling the definition of `foo` as a normal function definition. The compiler should be careful not to attempt tail recursion removal before opencoding, or it will make the primitive into small tight infinite loops! (In Common Lisp, the declaration `(declare (inline foo))` eliminates any possible confusion.)

5.3 Utah Common Lisp

Most Lisp systems are not designed to accommodate alternate representations. Implementors find it difficult to reconcile the necessary abstraction layer with stringent performance requirements, and abstraction generally loses the contest. The difficulty of abstraction goes up with the variety of representations desired; accommodating different tag assignments is trivial, but allowing tag, separate spaces, and BBOP representations is so difficult that it has not been completely accomplished by anyone.

The Utah Common Lisp (UCL) compiler is the latest in a series of efforts [84] to get good internal abstractions and good code quality at the same time. It is a fairly simple compiler, consisting of only four required phases, interleaved with optional optimization phases. The first required phase (*itemization*) converts S-expressions

into internal structures more suited for analysis, the second is the actual code generator producing assembly language with virtual register references, the third is a register (or resource) allocator, and the final phase is an assembler. Register allocation, assembly, and (to some extent) code generation are driven from the description of the target machine, while the function calling protocol is used by the code generator and register allocator, and only the code generator uses the function opencodings. Almost all optimization is in optional phases that transform code in some intermediate form to code in the same intermediate form, thus the compiler's functioning is not dependent on those phases. Without the optimizations, the basic UCL compiler is about 5000 lines of Common Lisp, and the 68000 machine description is about 1000 lines (other machine descriptions have not been completed, but are probably shorter, since the 68000 has a rather complicated instruction set). Figure 5.7 illustrates the compiler's structure.

5.3.1 Itemization

Itemization converts source code into *items*, which are structures of various types representing the different special forms of Common Lisp. This phase also does alpha-conversion of variables and functions, as well as expansion of macros.

Although the itemizer is basically independent of representation, a dependency has been introduced for the sake of expediency; instead of requiring fixnums to be created by calling a function at load time, the function `make-fixnum` transforms the number according to the chosen representation and writes the transformed value instead. Although this is convenient, there is an assumption that fixnums are represented as immediate rather than pointer objects, which constrains possible representation designs. This is also a problem for the automatic designer, since the specification cannot designate a particular type as fixnum, so the designer in turn cannot synthesize a definition for `make-fixnum`. It is to be hoped that later versions of the compiler will not include this function of doubtful benefit.

5.3.2 Code Generation

The primitive datatype implementations are used by the code generation phase. Since register allocation happens subsequently, the opencodings will use *virtual registers* (or *vregs*), of which it is assumed that there are an infinite number. It is possible to insist that certain vregs be allocated in certain register sets, or even to use registers directly, but this should be avoided, since the allocator is moderately intelligent about the right places for operands to be.

Since opencoding is just an alternate way of compiling a function call, a special operand `(arg` i`)` designates the ith argument, while `(result` i`)` designates the ith result (functions may return several values in Common Lisp[1]).

---

[1]For instance, bignums in Lucid Lisp [166] use multiple-valued opencoded functions in time-critical places.

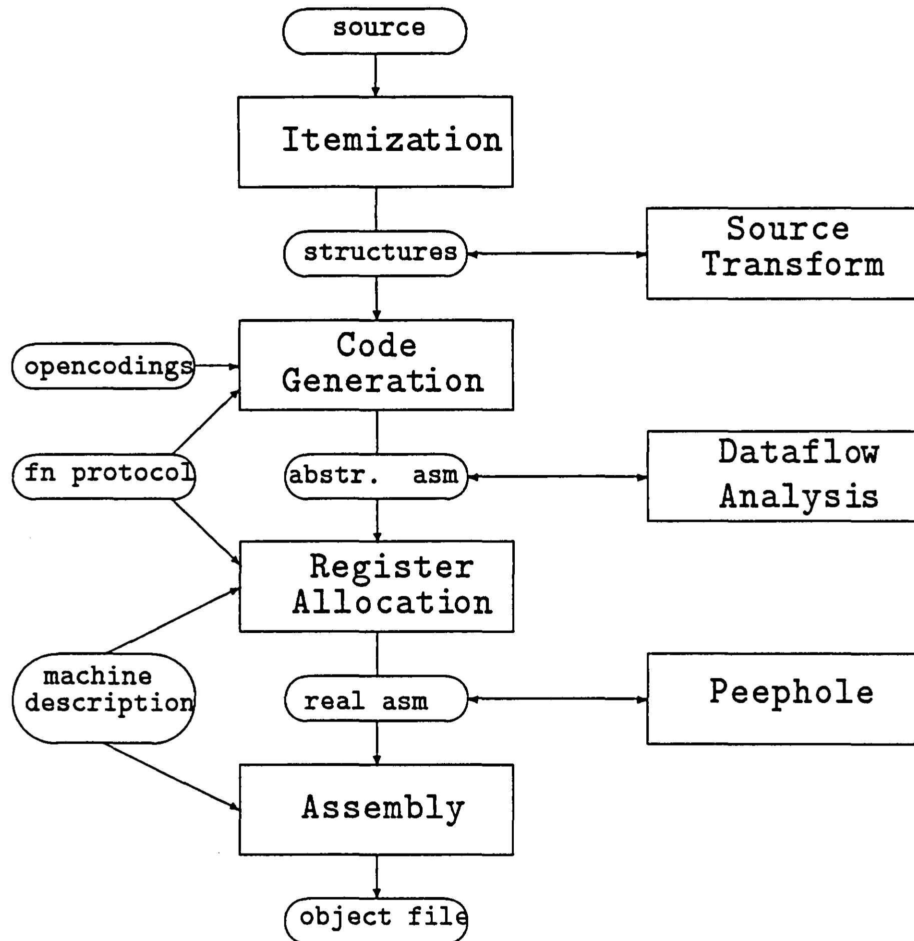

Figure 5.7. Architecture of the UCL Compiler

At the end of the code generation phase, the code is collected into basic blocks. Some dead code removal happens here, since unreachable code is not incorporated into any block and effectively disappears.

5.3.3 Register Allocation

The register allocator is a required phase that also does considerable optimization. It is driven from descriptions of the machine, of the function calling protocol, and from some general decisions about the desired usage of the machine's registers (such as dedicated registers). Since it works only on assembly language with virtual instead of real registers, it makes no assumptions about data representation.

5.3.4 Assembly

The assembler is totally driven from the machine description. In some ways, it resembles a Prolog interpreter, where the query is an instruction to be assembled, and the database is the machine description, since there are logical variables, a binding list, and backtracking. The assembler does not actually have any Lisp-specific information, so it does not depend on data representation. The output of the assembler is a *fasl file*, which is essentially object code, but includes commands to execute code while loading, and to create or manipulate symbols. The format is largely independent of representation, but it does assume that symbols are vector-like objects with small integer offsets, and that they are named by text characters. However, these symbols are required only to serve as convenient reference points, and they need not be used directly as the language system's symbols.

5.3.5 Micro-kernel

Strictly speaking, the micro-kernel is not part of the compiler, but it too must be free of any data representation assumptions. This is difficult, because the micro-kernel does provide the basic execution environment, and does have some characteristics imposed by the OS. At present, the micro-kernel is written in C.

The micro-kernel sets up some initial data areas: a Binary Program Space that is of fixed size, but which can be transformed into read-only space later on; a proto-symbol table which is dynamically allocated, and a proto-heap which is also dynamically allocated. Both the proto-symbol table and proto-heap need not be used permanently by the system, and in fact one stage of the normal UCL construction process transforms the symbol names from 0-terminated strings (C format) to character vectors with a length field (UCL format). Even so, the micro-kernel could not accommodate the Spice Lisp technique of dividing up the entire address space.

After the decision was made to use three low tag bits as the basic representation in UCL, some dependencies on this were introduced into the micro-kernel. The reason was to allow some of the C code to be used after the symbols had been transformed into the UCL format; a better approach might have been to load and use a Lisp version of the code instead, although this would make the bootstrapping process more complicated.

5.4 Evaluation

The designer and coder could be evaluated by counting operations, as was done in [141]. This method is error-prone, since the coder is fairly lax about optimality, and the cleanup happens in the UCL register allocator. Also, counting only allows conclusions about relative rather than absolute performance, and will fail to distinguish between important performance differences and insignificant ones, since overall execution time is not counted. Therefore, I shall concentrate on evaluation in a real system.

The complete evaluation method is somewhat involved. Basically, the designer and coder produce several files that are loaded into UCL running as a cross-compiler, whose output is then loaded and run by the micro-kernel. Figure 5.8 summarizes the process. Note that two files need to be cross-compiled: the definition of primitives generated by the designer, and the benchmark program itself. At the moment, the "boot file" does not make any direct references to data objects and need not be recompiled for different representations.

5.4.1 Benchmarks

The standard set of benchmarks for Lisp was assembled by Gabriel [53]. Despite their popularity, they have some serious disadvantages:

- Simplicity. Many of the benchmarks are simple enough that an analytical model could be developed, and a language designer could calculate provably optimal representations.
- Types. The programs require little or no dynamic polymorphism. A good type inferencer or a small set of declarations would eliminate any need for runtime type checking. There is a legitimate doubt, though, that real programs have much more dynamic polymorphism than do the benchmarks.[2]
- Variety. Even the larger benchmarks use only one or two types of objects. This distorts analyses by strongly favoring the one type in use and ignoring all others.

The larger benchmarks used by Steenkiste [150] and Shaw [137] are superior, since they are realistic programs with a greater complexity of data object usage. Unfortunately, they are also beyond the capabilities of the UCL system at present, and I am limited to the smaller Gabriel benchmarks.

5.4.2 Results

Designing implementations for simple s-expressions on the 68000 yielded several designs, which are summarized in Table 5.1. Each design was then loaded into the UCL compiler and used in compiling small benchmarks. The details of the process

[2]This is a complex issue in itself, and does not appear to have been studied satisfactorily.

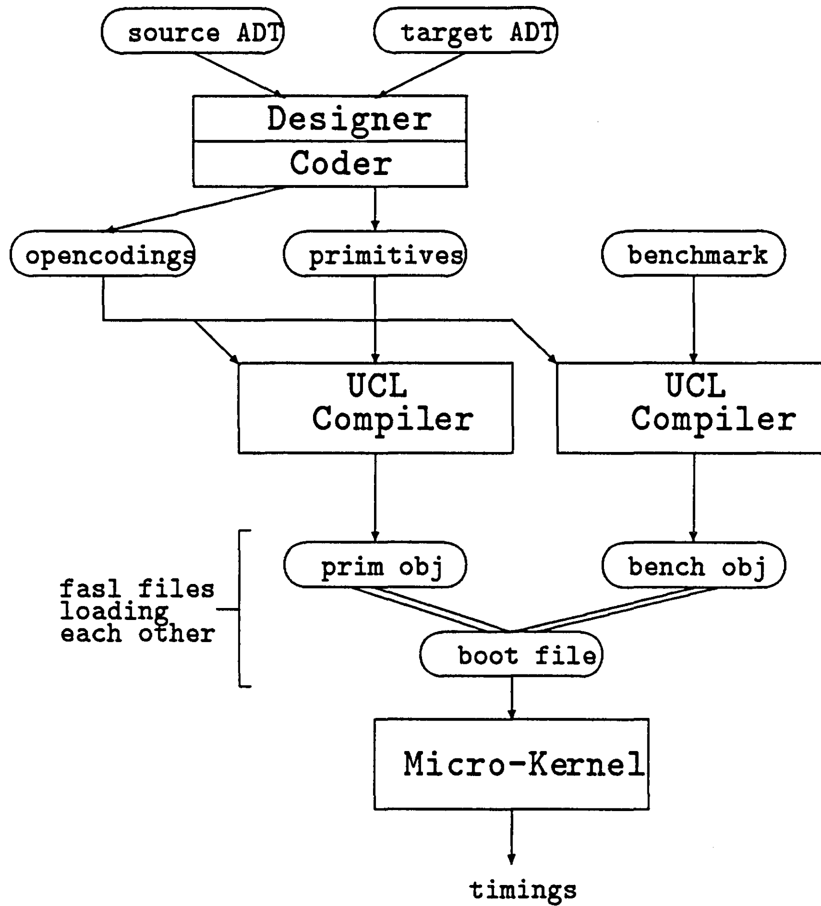

Figure 5.8. Evaluation Process

Number	Type	Tag Pos	0 Tag	First Slot
0	tags	high	integers	car
1	tags	high	lists	car
2	tags	low	integers	car
3	spaces	na	na	car
4	bbop	na	na	car

Table 5.1. Designs Produced

Program	Control	D0	D1	D2	D3	D4
tak	.23	.23	.33	??	.23	.23
takl	2.11	??	??	??	2.49	??

Table 5.2. Execution Times

may be found in Appendix B. To save time, several of the choice objects did not generate multiple alternatives, for instance the structure choice did not produce designs in which the cdr was the first slot of a list cell.

The first benchmark `tak` is a small highly recursive program that does integer arithmetic and comparison. `Takl` is a version of `tak` that uses lists of length n to represent the integer n. Times were measured in seconds of execution time on an HP 9000/350 workstation, a 25 MHz 68020-based machine. Since no garbage collection was performed, and the working set size was small, the execution times of each test case varied by less than 5%. The control case is the normal UCL compiler, which uses 3 low tag bits, although some of the compiled code seems to treat integers as untyped values.

The available numbers are summarized in Table 5.2. Several runs failed with coredumps or infinite loops, and debugging attempts never uncovered the reasons. There are several possibilities, including incorrect opencodings, incorrect cross-compilation, but most likely fatal interactions between the altered representations and the one that has been built into UCL. Since the testing environment is essentially raw machine code, it is almost impossible to determine the cause of failure. The numbers for `tak` are easy to interpret, since the only data operations are on small integers, and only D1 required any actual manipulation of a tag (note that the manipulation causes the benchmark to be 50% slower!), while the other designs operate on the integers directly.

5.5 Discussion

The actual experimental results are limited in quantity, and of doubtful usefulness, since the simulated results of [141] cast considerable doubt on the existence

of data representations that are good over a wide range of programs. However, the process of acquiring the results has been very informative as to areas that need further development. Each of these was resolved by taking a shortcut that should be addressed in the future:

- Design rules frequently assume machine-like targets.
- The designer encounters combinatorial explosions frequently, so it has restrictive rules and can only be used on simple types.
- The coder does not find and exploit puns by generating different code.
- The coder cannot allocate data to registers permanently.
- Coding frequently fails to match against machine instructions.
- Nothing is done about garbage collection or storage recovery in general, which means that allocation-intensive benchmarks cannot be run.

5.5.1 Lack of Generality

Although the interface to the designer appears rather general—it is specified to take two ADTs and produce an implementation of one in terms of the other, the reality is considerably less impressive. In fact, the designer is likely to fail when used for anything other than Lisp on contemporary hardware. The main limitation is in the rules, which will not match on unusual targets, ultimately causing the designer not to find any designs at all.

The right answer is a considerably more abstract and mathematical approach that can reason from the axioms themselves. However, this will exacerbate rather than alleviate the most serious problem of the designer, which is the rapid growth in the number of different designs, as the type to be implemented gets more complicated.

5.5.2 Combinatorial Explosion

As observed previously, the number of possible designs is huge, since there are usually several "opportunities" for combinatorial explosion. This means that a simple forward or backward chaining inference mechanism (as used both here and in [141]) is undesirable.

A better approach might be derived from the recent literature on expert systems in design (see Mostow's review [113] for work prior to 1985). Design is characterized as a task of describing an artifact that satisfies functional specifications including constraints on size, performance, etc. Clearly, the design of primitive datatype implementations fits this model; the functional specification consists of the abstract data type and the target machine, while constraints are time and space, on both the resulting design *and* on the designer itself (i.e. it cannot explore all alternatives).

After some time spent experimenting with versions of the more sophisticated algorithms for design, I found that the published systems were insufficiently adapted to the needs of datatype design.

The main problem with controlling the size and shape of the search space is that we have very little idea of how to determine which parts are uninteresting. In contrast to the design of physical objects, software design is highly "nonlinear"—in other words, small changes in the design may have a large effect on performance, while large changes may not have any effect at all. For instance, changing one bit of a 0 tag causes all operations to require tag operations, or incrementing a range of numbers by one may force storage allocation and reclamation. In these circumstances, design constraints are of little use, since all algorithms based on constraints assume some kind of continuity, that guides the algorithm toward a "nearby" better solution.

5.5.3 Coding Optimizations

Consider the use of a single low tag field on fixnums. The default wrapping will compose fixnum addition as a sequence of shifting the numbers down to remove the tag, adding them, then shifting up and adding the tag back in. In other words, given two tagged fixnums a and b, a tag field of n bits, a tag of t, and modelling the shifts as multiplication and division:

$$a \oplus b = (\lfloor a/2^n \rfloor + \lfloor b/2^n \rfloor) * 2^n + t$$

This formula does 3 shifts and 2 adds, which is unnecessary. Distributing the multiplication and using the definition of integer division yields

$$a \oplus b = a - a \bmod 2^n + b - b \bmod 2^n + t$$

Since $t = a \bmod 2^n$ and $t = b \bmod 2^n$, we can cancel:

$$a \oplus b = a + b - t$$

Thus, for any fixnum tag, addition can be reduced to the addition alone plus a subtraction of the tag. If the tag is zero, then no adjustment is needed at all. The derivation needs to be done for each individual primitive—subtraction requires an add of the tag, multiplication requires an extra shift, and so forth.

The reasoning here is moderately sophisticated and of sufficient difficulty that a term rewriting or computer algebra system would be appropriate. This is necessary to achieve human-quality primitives, since this is how low tag arithmetic functions are best written. An interesting piece of related work was done by Massalin [103], who set up lengthy searches for optimal machine language sequences for various arithmetic operations. This may prove useful for primitive functions as well.

5.5.4 Use of Registers

The coder does not know much about registers, mostly because the opencoding is assumed to happen before register allocation, so the coder produces references to virtual registers (temporary names).

However, some objects are sufficiently important that they should be permanently assigned to a fixed register. Some examples:

- Heavily-used objects such as `nil` and `0`.
- Bit patterns such as tag and data masks.
- Heavily-used addresses such as a pointer to the next free heap location.

The coder could handle this by noting any constant or variable that is referenced extensively, based on estimating expected uses (references in functions times the number of calls to the functions). The most-used things should go into registers, if enough are available. This decision interacts with other parts of the system, since for instance tying up a register with a constant means fewer available for allocation. This is an important tradeoff: the Cray-1 has dozens of registers while addressing memory is expensive (so each dedicated register has a measurable effect on performance [8]), while the Intel 80386 has only a few registers, all of which are needed as temporaries.

In UCL, it is possible to dedicate a register, but requires some extra work:

1. Add a declaration into the information used by the register allocator, so it will avoid using the register as temporary storage.
2. Add a load operation into the initialization code, ensuring that it comes after any prerequisite initializations. (A special cons cell cannot be created until a cons cell heap exists, for example.)
3. Add a declaration to some table that the compiler will use to decide how to compile references to that object (it should not go via the default). This table does not presently exist, and would require new compiler code.

5.6 Summary

The experimental work described here is rather limited in its extent, and could be expanded to answer a variety of questions. Although the machinery described makes accurate evaluation *possible*, it is by no means easy, since there are still issues of effects from the specification, and the validity of benchmark programs.

The designer program is limited because of combinatorial problems, while the coder is too simple to do justice to the more interesting design rules. Getting the entire language implementation to allow different designs is moderately difficult, because of the many places that dependencies creep in. This problem has already occurred in UCL, and constitutes a serious obstacle to further experimentation. Further efforts will be required to remove those dependencies.

CHAPTER 6

CONCLUSION

> The design of the execution environment of programs seems to be the subject of sporadic, disjointed research that is usually conducted as a part of some other activity.
>
> W.A. Wulf, *IEEE Computer* (1980)

Automatic design of primitive datatype representation can work. The progression, from formal definitions of datatypes and machines, to pieces of code incorporated into a real implementation, has been handled almost completely by a program. In order to do this, it was necessary to study past implementations, to arrive at a set of heuristics that could build designs in a reasonable amount of time, and to design a system for unbiased testing of different designs.

On the other hand, we should be careful not to overestimate this work. Many problems and issues have been glossed over, simplified, or ignored entirely. I am doubtful that all of these merely need patching up—some will require fundamental insights into the nature of implementation, others may be inherently insoluble. Even so, some extensions are promising, and there are potentially valuable applications for automatic design. Whether or not this particular work is a deadend, it may have uses in a somewhat different area, which will be described.

6.1 Contributions

Information about previous implementations has never before been gathered into one place and analyzed. There were no great surprises or overlooked geniuses, but this negative observation is itself useful information.

The formal model for implementations is not a particularly strong one, since it is essentially equivalent to general ADT models. Still, its use has revealed tacit assumptions in implementors' reasoning, as well as in Lisp language standards.

The heuristics for data structure design are somewhat more novel. Some have never been written down before, while others have more complete sets of preconditions than have been expressed previously. It has also become clear that adequate solutions to the problem require deeper reasoning than that offered by rules alone.

Experimentation with machine-generated datatype designs is new. It has been shown to work, but at the same time, the quality of the code is poor. For regular

use, the existing designer and coder would need considerable augmentation; less if they are intended only as a sort of "designer's assistant." The design-independent Lisp system is also a first; its importance in getting meaningful comparisons cannot be overemphasized.

Overall, the most important contribution of this work is the strong light that it casts on what had been a relatively unknown topic. In the future we can expect a more formal and systematic approach to the implementation of runtime systems of higher-level languages, leading perhaps to wider use of such languages and a corresponding increase in the quantity and quality of software in general.

6.1.1 Recommendations for Lisp Standards

The Common Lisp and Scheme standards need some additional parameters that describe the limits of implementations. For Common Lisp, the following should be defined:

max-integer-length The maximum number of bits appearing in any integer; therefore, it is an upper bound on the `integer-length` function. Can be `nil` if there are no limits short of memory availability.

random-state-length The number of bits in a random state. An alternate form is `max-random-period`, since the number of bits in a random state sets only the crudest bound on the period of the generator.

max-number-package-symbols The maximum number of symbols that can be in a package, either as internal or external symbols. Can be `nil` if there are no limits short of memory availability.

max-number-external-symbols The maximum number of symbols that can be external in a single package. Can be `nil` if there are no limits short of memory availability.

max-hash-table-size The maximum possible size for a hash table. Can be `nil` if there are no limits short of memory availability. Should be at least 1000.

max-structure-size The maximum number of slots allowed in a structure. (This one is dubious, since the syntax of `defstruct` makes $>$ 100 slot structures highly unlikely.)

These may be variables or constants defined for each system individually, or universal values that all conforming implementations can be expected to support. In addition, the standard should assert that properties of types carry over their uses, for instance, that symbol names inherit the 1000-character minimum imposed on strings.

Scheme as defined in [126] is a somewhat "looser" standard, as it seems to be intended to define common concepts and vocabulary, rather than to make promises about what a portable program can or cannot depend on. At the very least, a

Scheme system should define the allowable ranges of numbers (particularly integers) and the allowable sizes of strings and vectors. In addition, since a Scheme system is permitted to support only a small range of integers, the standard should state what will happen on integer arithmetic overflows.

Both Scheme and Common Lisp need characterizations for the memory space available. There are two possibilities, both of which could be used with either language.

One way is to use a version of the `sizeof` operator in C that would return the number of bits in an object or type, along with another function returning the number of bits currently available in the system. (Bits are really too fine-grained, but are also universal in a way that bytes, words, etc are not.)

(object-size *obj***)** Returns the minimum number of bits required for the general representation of the object.

(available-memory) Returns the number of bits available in memory.

There are some practical difficulties, since optimizations like cdr coding will require that consistent numbers assume the worst case, which may be much worse than reality. The effect would be to make some programs fail, even though enough space is available.

A better approach is somewhat more abstract, since it only counts objects:

(maximum-number *type***)** For any type specifier, returns the maximum number of this type of object that can be created. Returns `nil` if the number is unbounded; this will typically be the case only for fixnums, characters and other small immediate objects.

(number-left *type***)** For any type specifier, returns the number of objects that may be created before memory is exhausted. This may return different results after a reclamation.

Both approaches to calculating space have their problems, but experience with these would be necessary before a final assessment is possible.

6.2 Extensions

The existing designer and coder can be improved in many ways, most of which were mentioned in Chapter 5. There are also some broader extensions.

The current exhaustive algorithm is inadequate in realistic design spaces. Unless some means can be found to locate good designs before evaluation, the entire designer will be greatly limited in what it can do. One possibility is to change the shape of the design space, so that classes of designs can be handled as a unit.

The rules as they stand work only on the predefined schemas. Some of the work mentioned in the first chapter can derive implementations from arbitrary equations, but it is very limited at present. One attractive point of a more general and less heuristic designer is the possibility that it could discover new methods for

representation. This might be approached by having the machine build a language system and test it on benchmarks itself, then use the data so generated to alter its design heuristics. This approach has a real possibility for generating new knowledge about implementation, but it requires too much from the machine to be feasible at present.

Data type design is actually a subject of lesser significance; the design and implementation of control-related objects is much more important, but also more complicated in its demands on the available theory. Datatypes are simple because their formal specification involves a handful of simple axiom schemas; control structures involve the entire semantic definition of a language. There has been some work exploring this topic. Wand [161] has done some elegant development from language definition to closure-like structures built from combinators, using continuation semantics as an intermediate step. More pragmatic analyses by Biswas and Dasgupta [21] have compared various possibilities for stack structures in conventional languages. I should also mention that the formalism of this dissertation has a subtle bias towards heap-allocated objects (they are assumed to be independent of program scopes, for instance), but that eliminating this bias would also require knowing a lot about language semantics in general (see [114] for an example of using data flow information to deduce properties of list data structures).

Although I have not been directly concerned with parallelism, the importance of the topic demands some consideration. Parallel machines do offer some opportunities for optimization, for instance by requiring that datatypes be defined more abstractly, which in turn opens up new implementation possibilities. Many of the design rules will change, although this depends on the architecture. For message-passing and large-grained architectures, the rules will not be much different. Shared memory, however, can change the situation in at least two ways: the costs of getting to local vs remote memory must be taken into account, and the increased quantity of total memory may require more unusual addressing schemes. Perhaps the most significant tradeoff in data representation for parallelism is the choice between sharing and copying. Sharing uses less space, but slows things down by the serialization necessary to control indeterminacy. Copying, on the other hand, uses more space, and may still introduce problems with merging changed copies, if such becomes necessary. Issues like these point up the importance of overall language semantics; merging changed copies of data structures is not a major problem for a purely functional language.

6.3 Applications

One of the applications *not* mentioned in this section is the use of a designer on a regular basis. The prospects for a designer as a standard tool like Unix yacc are not very good. Even if the various problems were to be solved, the actual amount of effort saved by this system is relatively small. The effort saved is greater in those cases where extreme optimization is important, but this also requires more detailed data on usage statistics than is generally available, especially before a system has come into use. At the same time, the abstraction required in

the rest of the implementation is high, enough to overwhelm any savings from automatic design. Changing the designer to be a design-checking assistant might be worthwhile, particularly in the case of complex designs. This would eliminate combinatorial explosion as a problem, but the designer would also need a more elaborate human interface.

Another use for the design rules is as a way to explore the representation of critical data structures in normal programs. Evaluation would have to be altered to take the compiler into account. For instance, a frequently used structure in a C program could be analyzed to decide if bit encoding of fields would yield acceptable performance.

Other uses include the design of new hardware architectures. The degrees of freedom for a designer are much more numerous, resulting in a problem of knowing what to look for. Should an operation be done in hardware or software? If hardware can handle more kinds of representations, when does this impinge on chip real estate? The potential payoffs are very high, enough to justify significant amounts of supercomputer time searching the design space.

6.4 Abstract Data Types in General

Since this work is about implementation, there has been an underlying assumption that both the source language and the target machine have already been decided upon. A criticism of this basic viewpoint is that fitting the software to the language and machine is solving the wrong problem. Instead, either the machine must be fitted to the language, or the language designed to be readily implemented. The second point of view leads to languages like C, that are efficient to use, but offer little more than does assembly language, while the first leads to attempts to build "higher-level" machines, which have been tried many times, but have been almost completely unsuccessful.

A more subtle criticism has to do with the languages being implemented. Perhaps even Lisp and Prolog are too low-level; after all, many of the implementation and usage problems of Lisp ultimately derive from the possibility that circular lists may occur, and Prolog systems are complicated by the requirement to handle cuts at any point in a predicate. A higher-level language will make fewer compromises on such matters, perhaps simplifying the requirements on the implementation. For instance, EQLOG [57] is completely based on term rewriting in initial algebras defined by abstract data types. Although implementation by general term rewriting is very inefficient, programs are so abstract that dozens of wildly differing translations are possible. One important step in a translation process will be to select data representations that are best suited to a particular program. Instead of running for a long time to produce a fixed design for a language, the designer will be part of a compiler and produce designs adapted to specific programs. Although this is not a new idea, only limited success has been achieved so far. I believe that future progress will depend on a combination of techniques, including the ones described in this dissertation, on human assistance when necessary, and on the analysis of high-level programs to determine which techniques are the most useful.

Success could lead to a revolution in programming; the next several years will be interesting to watch.

APPENDIX A

SPECIFICATIONS OF TYPES

A.1 Basic Lisp Definition

```
;;;; Definition of datatypes for "pure Lisp"

(defadt purelisp
  (sum (atom smallints) (consp conses)))

(defadt conses
  (structure cons (car purelisp) (cdr purelisp))
  (maximum-number 1000000))

(defadt smallints
  (range 0 1000)
  (define fix+
    (lambda (x y)
      (declare (smallints x y))
      (int+ x y)))
  (define fix>=
    (lambda (x y)
      (declare (smallints x y))
      (int>= x y)))
  (maximum-number 100000))
```

A.2 Common Lisp Definition

```
;;;; Definition of CL (UCL version)

(defadt t+bottom
  (sum (tp cl) (bottomp bottom)))

(defadt bottom (set bottom))

(defadt cl
  (sum (numberp number)
       (symbolp symbol)
```

```
       (arrayp array)
       (characterp character)
       (consp cons)
       (null null)
       (packagep package)
       (hash-table-p hash-table)
       (random-state-p random-state)
       (readtablep readtable)
       (streamp stream))
)

(defadt number
  (sum (rationalp rational)
       (floatp float)
       (complexp complex)))

(defadt rational
  (sum (integerp integer)
       (ratiop ratio)))

(defadt integer (bits #.(expt 2 30)))  ; need better repn

(defadt ratio
  (structure make-ratio
             (numerator integer)
             (denominator integer)))

(defadt float
  (sum ;(short-float-p short-float)
       (single-float-p single-float)
       ;(long-float-p long-float)
       ;(double-float-p double-float)
  )
)

(defadt single-float
  (structure make-float (exponent exponent) (mantissa mantissa))
;  ieee-float
)

(defadt exponent (bits 7))

(defadt mantissa (bits 23))

;;; Complex numbers have components,
;;; but they are not required to be mutable.

(defadt complex
```

```
  (structure make-complex (realpart number) (imagpart number)))

(defadt symbol
  (structure make-symbol
             (symbol-name simple-string)
             (symbol-package package)
             (symbol-plist list set-symbol-plist)
             (symbol-value t+bottom set-symbol-value)
             (symbol-function function set-symbol-function))
  (define boundp
    (-> (symbol) (boolean))
    (lambda (x) (eq (symbol-value x) bottom)))
  (define fboundp
    (-> (symbol) (boolean))
    (lambda (x) (eq (symbol-function x) bottom)))
)

(defadt array
  (sum (general-array-p general-array)
       ; bit-vector
       (simple-string-p simple-string))
)

(defadt general-array (structure make-raw-array))

(defadt simple-string
  (vector make-string
          (0 1000 string-length)
          (schar character set-schar)))

(defadt character (range 0 256))

(defadt cons
  (structure cons
             (car t set-car)
             (cdr t set-cdr)))

(defadt null (range 0 1))

;;; Not really a primitive type in CL
;
;(defadt list
;  (sum cons null))

(defadt package
  (structure make-package
             (package-use-list list set-package-use-list)
             (package-internals symbol-vector set-package-internals)
```

```
            (package-externals symbol-vector set-package-externals)))

; hash-table, etc

(defadt hash-table (structure make-hash-table))

(defadt stream (structure make-stream))

(defadt readtable (structure make-readtable))

(defadt random-state (range 0 1000))
```

A.3 68000 Definition

```
;;;; Definition of 68000 (not 68020)

(defadt m68k (structure make-m68k (a areg) (d dreg) (m memory))
  (-> (setf ?d (lref (+ ?s ?i)))
      (move long (displacement ?s ?i) ?d))
  (-> (setf ?d (lref ?s))
      (move long (indirect ?s) ?d))
  (-> (setf ?d ?s)
      (move long ?s ?d))
  (-> (setf (lref (+ ?d ?i)) ?s)
      (move long ?s (displacement ?d ?i)))
  (-> (setf (lref ?d) ?s)
      (move long ?s (indirect ?d)))
  (-> (setf ?x (setf-lref ?d ?s))
      (move long ?s (indirect ?d))
      (move long ?d ?x))
  (-> (setf ?d (logand ?s1 ?s2))
      (move long ?s1 ?d)
      (and long ?s2 ?d))
  (-> (setf ?d (logior ?s1 ?s2))
      (move long ?s1 ?d)
; add only works because we only OR non-overlapping fields
;      (or long ?s2 ?d)
      (add long ?s2 ?d))
  (-> (setf ?d (int+ ?s1 ?s2))
      (move long ?s1 ?d)
      (add long ?s2 ?d))
  (-> (setf ?d (int- ?s1 ?s2))
      (move long ?s1 ?d)
      (sub long ?s2 ?d))
  (-> (setf ?d (ash ?s 1))
      (add long ?s ?s))
  (-> (setf ?d (ash ?s -1))
      (asr long ?s ?d))
```

```
  (-> (setf ?d (eq ?s1 ?s2))
      (cmp long ?s1 ?s2)
      (%cjump ne (label false)))
  (-> (setf ?d (int>= ?s1 ?s2))
      (move long ?s1 (temp d 1))
      (%cjump ?s2 (temp d 1) lt (label false)))
  (-> (setf nil (int>= ?s1 ?s2))
      (move long ?s1 (temp d 1))
      (%cjump ?s2 (temp d 1) lt (label false)))

  (define lref
    (-> (integers) (longword))
    (lambda (x) (mref (m *machine*) x)))
)

(defadt areg
  (vector make-areg 8 (a-ref address)))

(defadt dreg
  (vector make-dreg 8 (d-ref longword)))

(defadt memory
  (vector make-memory 16000000 (mref byte setf-mref)))

(defadt byte (bits 8))

(defadt address (bits 24))

(defadt longword (bits 32))

;;; Miscellaneous definitions (all are kludges).

(defun machine-word-size (m) 32)

(defun machine-word-accessor (m) 'lref)

(defun machine-word-setter (m) 'setf-lref)

;;; To decide what is main memory, look for a vector with >500 elements?

(defun vector-accessor (adt)
  (car (fourth (adt-schema adt))))

(defun memory (adt)
  (find-adt 'memory))
```

APPENDIX B

COMPLETE DESIGNER RUN

B.1 Designer Session

```
                    Common Lisp

            Part No. 98678A   Rev. 1.01
(c) Copyright 1986, Hewlett-Packard Company.  All rights reserved.

UCL Cross Compiler,  4-Jul-88

;;; This Lisp is already set up as a cross-compiler, but we can still
;;; load the data structure designer and coder.

(load "dsd")
!!! Warning: No declaration
!!! Warning: No declaration
!!! Warning: No declaration
!!! Warning: No declaration
!!! Warning: No declaration
!!! Warning: No declaration
!!! Warning: No declaration
!!! Warning: No declaration
!!! Warning: No declaration
!!! Warning: No declaration
!!! Warning: No declaration
!!! Warning: BYTE already defined
"/u/shebs/kbi/dsd.l"

;;; A number of ADT definitions are already loaded, including those for
;;; simple S-expressions and for the 68000, so can go ahead and start
;;; designing.

;;; Each choice announces its success (though not on which design),
;;; and the lists separated by ellipses are lists of successful and
;;; untried goals after each choice has executed.

;;; When all the designs have been found, the coder starts work.
```

```
(impl 'purelisp 'm68k)
NIL ... NIL
Choice REALITY-CHECK succeeded...
(FEASIBLE) ... (PURELISP)
Making decision (PURELISP . BBOP)...
Choice BBOP-SUM succeeded...
(PURELISP FEASIBLE) ... (CONSES SMALLINTS)
Choice BLOCK-STRUCTURE succeeded...
(CONSES PURELISP FEASIBLE) ... (SMALLINTS)
Choice DIRECT-RANGE succeeded...
(SMALLINTS CONSES PURELISP FEASIBLE) ... NIL
Making decision (PURELISP . SPACES)...
Choice SPACED-SUM succeeded...
(PURELISP FEASIBLE) ... (CONSES SMALLINTS)
Choice BLOCK-STRUCTURE succeeded...
(CONSES PURELISP FEASIBLE) ... (SMALLINTS)
Choice DIRECT-RANGE succeeded...
(SMALLINTS CONSES PURELISP FEASIBLE) ... NIL
Making decision (PURELISP . TAGGED)...
Choice LO-TAGGED-SUM succeeded...
(PURELISP FEASIBLE) ... (CONSES SMALLINTS)
Choice BLOCK-STRUCTURE succeeded...
(CONSES PURELISP FEASIBLE) ... (SMALLINTS)
Choice DIRECT-RANGE succeeded...
(SMALLINTS CONSES PURELISP FEASIBLE) ... NIL
Making decision (PURELISP TAGGED 0 1)...
Making decision (PURELISP TAGGED 1 0)...
Choice HI-TAGGED-SUM succeeded...
(PURELISP FEASIBLE) ... (CONSES SMALLINTS)
Choice BLOCK-STRUCTURE succeeded...
(CONSES PURELISP FEASIBLE) ... (SMALLINTS)
Choice DIRECT-RANGE succeeded...
(SMALLINTS CONSES PURELISP FEASIBLE) ... NIL
(PURELISP FEASIBLE) ... (CONSES SMALLINTS)
Choice BLOCK-STRUCTURE succeeded...
(CONSES PURELISP FEASIBLE) ... (SMALLINTS)
Choice DIRECT-RANGE succeeded...
(SMALLINTS CONSES PURELISP FEASIBLE) ... NIL
5 designs found.
!!! Warning: No meaning for NEXT-FREE-CONSES
!!! Warning: No meaning for NEXT-FREE-CONSES
!!! Warning: No meaning for NEXT-FREE-CONSES
;;; GC starting
;;; GC 4: time 12400 milliseconds
;;; GC 160714 stable, 69798 active, 574054 recovered, 584031 free
!!! Warning: No meaning for NEXT-FREE-CONSES
!!! Warning: No meaning for NEXT-FREE-CONSES
```

```
!!! Warning: No meaning for NEXT-FREE-CONSES
!!! Warning: No meaning for NEXT-FREE-CONSES
!!! Warning: No meaning for NEXT-FREE-CONSES
!!! Warning: No meaning for NEXT-FREE-CONSES
;;; GC starting
;;; GC 5: time 10460 milliseconds
;;; GC 24636 stable, 210174 active, 569738 recovered, 579733 free
!!! Warning: No meaning for NIL
!!! Warning: No meaning for NEXT-FREE-CONSES
!!! Warning: No meaning for NEXT-FREE-CONSES
!!! Warning: No meaning for NEXT-FREE-CONSES
NIL

;;; The file c0.l contains opencodings for cross-compilation.  Loading it
;;; causes some opencodings to be redefined, leaving others as they were
;;; originally (there are a lot of opencodings, many specific to the
;;; machine but not to a particular data representation).

(load "c0.l")
"c0.l"

;;; Now ready to compile both the benchmark...

(ucf "tak.l")
Compiling "tak.l" to "tak.b" ...
RUN-BENCHMARK TAK
Compilation of "tak.l" complete.
T

;;; and the primitive function definitions.

(ucf "p0.l")
Compiling "p0.l" to "p0.b" ...
INIT-PURELISP FIX+ FIX>= ALLOCATE-CONSES
!!! Warning: Using NEXT-FREE-CONSES as Special
!!! Warning: Using NEXT-FREE-CONSES as Special
CONS
;;; GC starting
;;; GC 6: time 11560 milliseconds
;;; GC 234572 stable, 8194 active, 561800 recovered, 571777 free
T2 CDR T1 CAR CONSP ATOM
Compilation of "p0.l" complete.
T

;;; Lisp system's work is now done.
```

B.2 Abstract Design 0

```
;;; Design 0 for purelisp on m68k
;;; Decisions:
((purelisp tagged 0 1) (conses . block) (smallints . direct))

(define
 init-purelisp
 (lambda ()
  (let ((space (allocate-proto-heap 10000)))
    (setq heap-lower-bound space)
    (setq heap-upper-bound (fix+ space 10000))
    (setq next-free-heap space))))

(define fix+ (lambda (x y) (int+ x y)))

(define fix>= (lambda (x y) (int>= x y)))

(define
 allocate-conses
 (lambda ()
  (let ((new next-free-conses))
    (setq next-free-conses (fix+ next-free-conses 8))
    new)))

(define
 cons
 (lambda (t11 t12)
  (let ((new (allocate-conses))) (t1 new t11) (t2 new t12) new)))

(define t2 (lambda (x v) (setf-lref (int+ x 1) v)))

(define cdr (lambda (x) (lref (int+ x 1))))

(define t1 (lambda (x v) (setf-lref (int+ x 0) v)))

(define car (lambda (x) (lref (int+ x 0))))

(define next-free-heap nil)

(define heap-upper-bound nil)

(define heap-lower-bound nil)

(define consp (lambda (x) (eq (logand x 2147483648) 0)))

(define atom (lambda (x) (eq (logand x 2147483648) 0)))
```

B.3 Opencodings for Design 0

```
;;;; Machine-generated opencodings

(defopen fix+
        (move long (arg 0) (temp * 1))
        (move long (arg 1) (temp * 0))
        (move long (temp * 1) (result 0))
        (add long (temp * 0) (result 0)))

(progn
  (defopenp fix>=
           jumpnil
           (move long (arg 0) (temp * 1))
           (move long (arg 1) (temp * 0))
           (move long (temp * 1) (temp d 1))
           (%cjump (temp * 0) (temp d 1) lt (label false)))
  (defopenp fix>=
           jumpt
           (move long (arg 0) (temp * 1))
           (move long (arg 1) (temp * 0))
           (move long (temp * 1) (temp d 1))
           (%cjump (temp * 0) (temp d 1) ge (label false))))

(defopen t2
        (move long (arg 0) (temp * 3))
        (move long (immediate long 1) (temp * 2))
        (move long (temp * 3) (temp * 1))
        (add long (temp * 2) (temp * 1))
        (move long (arg 1) (temp * 0))
        (move long (temp * 0) (indirect (temp * 1)))
        (move long (temp * 1) (result 0)))

(defopen cdr
        (move long (arg 0) (temp * 2))
        (move long (immediate long 1) (temp * 1))
        (move long (temp * 2) (temp * 0))
        (add long (temp * 1) (temp * 0))
        (move long (indirect (temp * 0)) (result 0)))

(defopen t1
        (move long (arg 0) (temp * 3))
        (move long (immediate long 0) (temp * 2))
        (move long (temp * 3) (temp * 1))
        (add long (temp * 2) (temp * 1))
        (move long (arg 1) (temp * 0))
        (move long (temp * 0) (indirect (temp * 1)))
        (move long (temp * 1) (result 0)))
```

```
(defopen car
        (move long (arg 0) (temp * 2))
        (move long (immediate long 0) (temp * 1))
        (move long (temp * 2) (temp * 0))
        (add long (temp * 1) (temp * 0))
        (move long (indirect (temp * 0)) (result 0)))

(defsysvar next-free-heap)

(defsysvar heap-upper-bound)

(defsysvar heap-lower-bound)
```

B.4 Primitives for Design 0

```
;;;; Machine-generated primitives

(defun init-purelisp ()
 (let ((space (allocate-proto-heap 10000)))
   (setq heap-lower-bound space)
   (setq heap-upper-bound (fix+ space 10000))
   (setq next-free-heap space)))

(defun fix+ (x y) (declare (inline fix+)) (fix+ x y))

(defun fix>= (x y) (declare (inline fix>=)) (if (fix>= x y) t nil))

(defun allocate-conses ()
 (let ((new next-free-conses))
   (setq next-free-conses (fix+ next-free-conses 8))
   new))

(defun cons (t11 t12)
 (let ((new (allocate-conses))) (t1 new t11) (t2 new t12) new))

(defun t2 (x y) (declare (inline t2)) (t2 x y))

(defun cdr (x) (declare (inline cdr)) (cdr x))

(defun t1 (x y) (declare (inline t1)) (t1 x y))

(defun car (x) (declare (inline car)) (car x))

(defun consp (x) (eq (logand x 2147483648) 0))

(defun atom (x) (eq (logand x 2147483648) 0))
```

B.5 Bootstrap File

The bootstrap file `boot.l`, when compiled to `boot.b`, will be automatically loaded by the micro-kernel when it starts execution. When the file is loaded, the symbol `boot-system` is searched for and its code executed. In this case, the symbol's code loads more files and eventually executes `run-benchmark`, which should run the benchmark itself.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; File:         boot9.l
; Description:
; Author:       Leigh Stoller
; Created:      19-Mar-88
; Package:
; RCS $Header: boot9.l,v 1.1 88/04/02 20:56:49 stoller Exp $
;
; (c) Copyright 1988, University of Utah, all rights reserved.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; This must always be the first function defined. It redefines the C
;; version so that we can figure (by opencode) what function we were
;; trying to call. It is very dependent on the calling model, and must
;; looked at if we drop the frame pointer. Further, because we cannot
;; define strings yet, use a *large* symbol for the error message.

(defun undefined-function ()
  (declare (inline get-undefined-function-symbol))
  (console-print-string (symbol-name '|Undefined function called: |))
  (console-print-symbol (get-undefined-function-symbol))
  (console-print-newline)
  (exit-to-os -1))

(defun boot-system ()
  (fasl-load (symbol-name '|prims.b|))
  (init-purelisp)
  (fasl-load (symbol-name '|bench.b|))
  (run-benchmark))
```

B.6 Benchmark Run

```
% mv tak.b bench.b
% mv p0.b prims.b
% kernel
Time elasped: 0.230000 user, 0.000000 system

7
%
```

REFERENCES

[1] Abrahams, P. LISP 2 interim programming manual. Tech Memo TM-2710/111/00, System Development Corp., Jan. 1966.

[2] Abrahams, P., Barnett, J., Book, E., Firth, D., Kameny, S., Weissman, C., Hawkinson, L., Levin, M., and Saunders, R. The LISP 2 programming language and system. In *AFIPS Fall Joint Computer Conference* (1966), pp. 661–676.

[3] *Military standard: Ada programming language.* Department of Defense, Washington, D. C. 20301, 1980. MIL-STD-1815.

[4] Aho, A., Sethi, R., and Ullman, J. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[5] Alberga, C. N., Bosman-Clark, C., Mikelsons, M., Deusen, M. S. V., and Padget, J. Experience with an uncommon Lisp. In *Proc. 1986 ACM Conference on Lisp and Functional Programming* (Cambridge MA, Aug. 1986), ACM SIGPLAN/SIGACT/SIGART, pp. 39–53.

[6] Allen, J. R. *The Anatomy of LISP.* McGraw-Hill, 1978.

[7] American National Standards Institute. *ANSI C Standard.* 1986.

[8] Anderson, J. W., Galway, W. H., Kessler, R. R., Melenk, H., and Neun, W. Implementing and optimizing lisp for the cray. *IEEE Software* (July 1987), 74–83.

[9] Arvind, and Culler, D. E. Dataflow architectures. In *Annual Review in Computer Science* (Palo Alto CA, 1986), Annual Reviews Inc., pp. 225–254.

[10] Backus, J. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Commun. ACM 21*, 8 (Aug. 1979), 613–641.

[11] Backus, J., et al. The Fortran automatic coding system. In *Western Joint Computer Conference* (1957), pp. 188–198.

[12] Barbacci, M. Instruction Set Processor Specifications (ISPS): the notation and its applications. *IEEE Transactions on Computers C-30*, 1 (Jan. 1981), 24–40.

[13] Barr, A., and Feigenbaum, E., Eds. *The Handbook of Artificial Intelligence.* Vol. 1, William Kaufmann, Inc., Los Altos CA, 1982.

[14] Barron, D. *Pascal—The Language and its Implementation.* Wiley, Chichester, 1981.

[15] Barstow, D. *Knowledge-Based Program Construction.* North Holland, 1977.

[16] Bartley, D. H., and Jensen, J. C. The implementation of PC Scheme. In *Proc. 1986 ACM Conference on Lisp and Functional Programming* (Cambridge MA, Aug. 1986), ACM SIGPLAN/SIGACT/SIGART, pp. 86–93.

[17] Bates, R. L., Dyer, D., and Kooman, J. A. G. M. Implementation of Interlisp on the VAX. In *Proc. 1982 ACM Symposium on LISP and Functional Programming* (Pittsburgh PA, Aug. 1982), ACM SIGPLAN/SIGACT/SIGART, pp. 81–87.

[18] Bawden, A., Greenblatt, R., Holloway, J., Knight, T., Moon, D., and Weinreb, D. Lisp machine progress report. AI Memo 444, MIT AI Lab, August 1977.

[19] Bell, C., and Newell, A. *Computer Structures: Readings and Examples.* McGraw-Hill, New York NY, 1971.

[20] Berkeley, E., and Bobrow, D., Eds. *The Programming Language LISP: Its Operation and Applications.* Information International, Inc., 200 6th St., Cambridge MA 02142, 1964.

[21] Biswas, P., and Dasgupta, S. Architectural support for variable addressing in Ada—a design approach. *International Journal of Computer and Information Sciences 14*, 1 (Feb. 1985), 51–72.

[22] Bobrow, D., and Clark, D. Compact encodings of list structure. *ACM Transactions on Programming Languages and Systems 1*, 2 (Oct. 1979), 266–286.

[23] Bobrow, D., and Murphy, D. The structure of a LISP system using two-level storage. *Commun. ACM 10*, 3 (March 1967), 155–159.

[24] Brooks, R. A., Gabriel, R. P., , and Steele Jr., G. L. An optimizing compiler for lexically scoped lisp. In *Proceedings of the SIGPLAN 82 Symposium on Compiler Construction* (Boston MA, June 1982), ACM SIGPLAN, pp. 261–275.

[25] Brooks, R. A., Gabriel, R. P., and Steele Jr., G. L. S-1 Common Lisp implementation. In *Proc. 1982 ACM Symposium on LISP and Functional Programming* (Pittsburgh PA, Aug. 1982), ACM SIGPLAN/SIGACT/SIGART, pp. 108–113.

[26] Budd, T. *A Little Smalltalk.* Addison-Wesley, 1987.

[27] Campbell, J., and Hardy, S. Should Prolog be list or record oriented? In *Implementations of Prolog* (1984), J. Campbell, Ed., Ellis Horwood, pp. 369–375.

[28] Cardelli, L., and Wegner, P. On understanding types, data abstraction, and polymorphism. *Computing Surveys 17*, 4 (Dec. 1985).

[29] Cartwright, R. A constructive alternative to axiomatic data type definitions. In *Proc. 1980 LISP Conference* (1980), pp. 46–55.

[30] Cattell, R. Automatic derivation of code generators from machine descriptions. *ACM Transactions on Programming Languages and Systems 2*, 2 (Apr. 1980), 173–190.

[31] Caudill, P. A third generation Smalltalk-80 implementation. In *Object-Oriented Programming Systems, Languages, and Applications 1986 Conference Proceedings* (Portland OR, Oct. 1986), ACM SIGPLAN, pp. 119–130.

[32] Chailloux, J., Devin, M., and Hullot, J. LE LISP, a portable and efficient LISP system. In *Proc. 1984 ACM Symposium on LISP and Functional Programming* (Austin TX, Aug. 1984), ACM SIGPLAN/SIGACT/SIGART, pp. 113–122.

[33] Clark, D. W., and Green, C. An empirical study of list structure in Lisp. *Commun. ACM 20*, 2 (Feb. 1977), 78–87.

[34] Cohen, J. Garbage collection of linked data structures. *Computing Surveys 13*, 3 (Sep. 1981), 341–367.

[35] Computation Center, The University of Texas at Austin. *LISP Reference Manual.* 1975.

[36] Darlington, J. An experimental program transformation and synthesis system. *Artificial Intelligence 16* (1981), 1–46.

[37] Davidson, J., and Fraser, C. The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems 2*, 2 (Apr. 1980), 191–202.

[38] Davidson, J. W., and Fraser, C. W. Automatic generation of peephole optimizations. In *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction* (Montreal, Canada, 1984), ACM SIGPLAN, pp. 111–116.

[39] Debray, S., and Warren, D. SB-Prolog. Tech. Rep., SUNY Dept. of Computer Science, 1987.

[40] Digital Equipment Corporation. *VAX Architecture Handbook.* 1981.

[41] Deutsch, L. ByteLisp and its Alto implementation. In *Proc. 1980 LISP Conference* (Stanford CA, 1980), pp. 231–242.

[42] Deutsch, L. A LISP machine with very compact programs. In *Proceedings of IJCAI* (1973), pp. 697–703.

[43] Deutsch, L. P., and Bobrow, D. G. An efficient, incremental, automatic garbage collector. *Commun. ACM 19*, 10 (October 1976), 522–526.

[44] Ellis, J. *Bulldog: a Compiler for VLIW Architectures.* MIT Press, 1986.

[45] Fateman, R. Reply to an editorial. *ACM SIGSAM Bulletin 25* (March 1973), 9–11.

[46] Fitch, J., and Norman, A. Implementing LISP in a high level language. *Software—Practice and Experience 7* (1977), 713–737.

[47] Fladung, B. *The XLISP Primer.* Prentice-Hall, 1987.

[48] Foderaro, J. K., and Sklower, K. L. *The Franz Lisp Manual.* Berkeley CA, September 1981.

[49] UC Berkeley. *Berkeley FP User's Manual, Rev. 4.1.* 1983.

[50] Fraser, C. W. A knowledge-based code generator generator. In *Proc. Symp. on Artifical Intelligence and Programming Languages* (Rochester NY, Aug. 1977), ACM SIGART/SIGPLAN, pp. 126–129.

[51] Freudenberger, S., Schwartz, J., and Sharir, M. Experience with the SETL optimizer. *ACM Transactions on Programming Languages and Systems 5*, 1 (Jan.. 1983), 26–45.

[52] Friedman, D., Haynes, C., Kohlbecker, E., and Wand, M. Scheme 84 reference manual. Tech. Rep. 153, Computer Science Department, Indiana University, February 1984.

[53] Gabriel, R. *Performance and Evaluation of Lisp Systems.* MIT Press, 1985.

[54] Ganapathi, M., and Fischer, C. Affix grammar driven code generation. *ACM Transactions on Programming Languages and Systems 7*, 4 (Oct. 1985), 560–599.

[55] Gelernter, H., Hansen, J., and Gerberich, C. A compiled Fortran list processing language. *Journal of the ACM 7* (1960), 87–101.

[56] Glanville, R. S., and Graham, S. L. A new method for compiler code generation. In *Conf. Rec. of the Fifth Annual Symposium on Principles of Programming Languages* (January 1978), ACM SIGACT/SIGPLAN, p. 231.

[57] Goguen, J., and Meseguer, J. EQLOG: equality, types and generic modules for logic programming. In *Logic Programming: Functions, Relations, and Equations* (1986), Prentice-Hall, pp. 295–363.

[58] Goldberg, A., and Robson, D. *Smalltalk-80: the Language and Its Implementation.* Addison-Wesley, 1983.

[59] Gotlieb, C., and Tompa, F. Choosing a storage schema. *Acta Informatica 3* (1974), 297–319.

[60] Goto, E., Soma, T., Inada, N., Ida, T., Idesawa, M., Hiraki, K., Suzuki, M., Shimizu, K., and Philipov, B. Design of a Lisp machine - FLATS. In *Proc. 1982 ACM Symposium on LISP and Functional Programming* (Pittsburgh PA, Aug. 1982), ACM SIGPLAN/SIGACT/SIGART.

[61] Gries, D. *Compiler Construction for Digital Computers.* John Wiley and Sons, Inc., New York NY, 1971.

[62] Griss, M. L. A portable implementation of standard LISP. Utah Symbolic Computation Group Opnote No. 47, University of Utah, Department of Computer Science, August 1980.

[63] Griss, M. L., Benson, E., and Maguire Jr., G. Q. PSL: a portable LISP system. In *Proc. 1982 ACM Symposium on LISP and Functional Programming* (Pittsburgh PA, Aug. 1982), ACM SIGPLAN/SIGACT/SIGART, pp. 88–97.

[64] Griss, M. L., and Hearn, A. C. A portable LISP compiler. *Software—Practice and Experience 11* (June 1981), 541–605.

[65] Griss, M. L., and others. PSL implementation guide. Utah Symbolic Computation Group, University of Utah, Department of Computer Science, May 1982.

[66] Griswold, R. *The Macro Implementation of SNOBOL4.* W.H. Freeman, 1972.

[67] Griswold, R., and Griswold, M. *The Implementation of the Icon Programming Language.* Princeton University Press, 1986.

[68] Hellerman, H. Addressing multidimensional arrays. *Commun. ACM 5* (1962), 205–207.

[69] Hennessy, J., and Ganapathi, M. Advances in compiler technology. In *Annual Review in Computer Science* (Palo Alto CA, 1986), Annual Reviews Inc., pp. 83–106.

[70] Hewlett-Packard. *LISP Programmer's Guide.* 1986.

[71] Hill, F., and Peterson, G. *Digital Systems: Hardware Organization and Design, 2nd ed.* John Wiley and Sons, Inc., 1978.

[72] Hisgen, A., Lamb, D. A., Rosenberg, J., and Sherman, M. A runtime representation for Ada variables and types. In *Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language* (Dec. 1980), pp. 82–90.

[73] Huet, G., and Oppen, D. C. Equations and rewrite rules: a survey. In *Formal Language Theory: Perspectives and Open Problems* (1980), R. V. Book, Ed., Academic Press.

[74] Jalote, P. Synthesizing implementations of abstract data types from axiomatic specifications. *Software—Practice and Experience 17*, 11 (Nov. 1987), 847–858.

[75] Jenkins, M. Q'Nial: a portable interpreter for the nested interactive array language, Nial. Tech. Rep. 87-202, Queen's University, Dec. 1987.

[76] Jenkins, M., and Jenkins, W. *The Q'Nial Reference Manual.* Nial Systems Ltd., 1985.

[77] Kaehler, T. Virtual memory on a narrow machine for an object-oriented language. In *Object-Oriented Programming Systems, Languages, and Applications 1986 Conference Proceedings* (Portland OR, Oct. 1986), ACM SIGPLAN, pp. 87–106.

[78] Kahn, G., MacQueen, D., and Plotkin, G. *Semantics of Data Types. Lecture Notes in Computer Science*, Springer-Verlag, 1984.

[79] Kaneko, K., and Yuasa, K. A new implementation technique for the UtiLisp system. In *Proceedings of the SIGSYM Meeting* (June 1987), Information Processing Society of Japan, pp. 1–7.

[80] Kant, E. The selection of efficient implementations for a high-level language. In *Proc. Symp. on Artifical Intelligence and Programming Languages* (Rochester NY, Aug. 1977), ACM SIGART/SIGPLAN, pp. 140–146.

[81] Kapur, D., and Srivas, M. A rewrite rule based approach for synthesizing abstract data types. In *Mathematical Foundations of Software Development*, Springer-Verlag, 1985, pp. 188–207.

[82] Kessler, R. R. *COG: An Architectural-Description-Driven Compiler Generator.* PhD thesis, Department of Computer Science, University of Utah, Salt Lake City, Utah 84112, January 1981.

[83] Kessler, R. R. Peep - an architectural description driven peephole optimizer. In *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction* (Montreal, Canada, 1984), ACM SIGPLAN, pp. 106–110.

[84] Kessler, R. R., Peterson, J. C., Carr, H., Duggan, G. P., Knell, J., and Krohnfeldt, J. J. EPIC - a retargetable, highly optimizing Lisp compiler. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction* (1986), pp. 118–130.

[85] Kirby, R. ULISP for PDP-11s with memory management. Tech. Rep. TR-546, Computer Science Center, University of Maryland, June 1977.

[86] Knuth, D. *The Art of Computer Programming: Fundamental Algorithms*, 2 ed. Vol. 1, Addison-Wesley, 1981.

[87] Knuth, D. *The Art of Computer Programming: Seminumerical Algorithms*, 2 ed. Vol. 2, Addison-Wesley, 1981.

[88] Knuth, D. *The Art of Computer Programming: Sorting and Searching.* Vol. 3, Addison-Wesley, 1981.

[89] Kodasky, J. LISP-11. Tech. Rep., DECUS, Jan. 1977.

[90] Krasner, G., Ed. *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley, 1983.

[91] Landin, P. J. The next 700 programming languages. *Commun. ACM 9*, 3 (March 1966), 157–164.

[92] Lecarme, O., and Gart, M. *Software Portability.* McGraw-Hill, 1986.

[93] Lenat, D. Eurisko: a program that learns new heuristics and domain concepts. *Artificial Intelligence 21* (1983), 61–98.

[94] Leverett, B. W., Cattell, R. G. G., Hobbs, S. O., Newcomer, J. M., Reiner, A. H., Schaltz, B. R., and Wulf, W. A. An overview of the production quality compiler-compiler project. *IEEE Computer 13*, 8 (August 1980), 38–49.

[95] Levin, M. I., and Berkeley, E. C. LISP 2 primer. Tech Memo TM-2710/101/00, System Development Corp., July 1966.

[96] Lewis, D., Galloway, D., Francis, R., and Thomson, B. Swamp: a fast processor for Smalltalk-80. In *Object-Oriented Programming Systems, Languages, and Applications 1986 Conference Proceedings* (Portland OR, Oct. 1986), ACM SIGPLAN, pp. 131–139.

[97] Lieberman, H., and Hewitt, C. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM 26*, 6 (June 1983), 419–429.

[98] Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, C., Scheifler, R., and Snyder, A. CLU reference manual. Tech. Rep. TR-225, MIT Laboratory for Computer Science, Oct. 1979.

[99] Loosemore, S. J. A-LISP: a small Lisp implementation. 1988. Manual and implementation details.

[100] Low, J. R. Data structure selection: an example and overview. *Commun. ACM 21*, 5 (May 1978), 376–385.

[101] Malcolm, M. A., and Stafford, G. J. The Thoth assembler writing kit. Tech. Rep. CS-77-14, University of Waterloo Dept. of Computer Science, September 1977.

[102] Marti, J. B., Hearn, A. C., Griss, M. L., and Griss, C. Standard LISP report. *SIGPLAN Notices 14*, 10 (October 1979), 48–68.

[103] Massalin, H. Superoptimizer: a look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct. 1987), ACM SIGARCH/SIGPLAN/SIGOPS, pp. 122–126.

[104] Matula, D., and Kornerup, P. Finite precision rational arithmetic: slash number systems. *IEEE Transactions on Computers C-34*, 1 (Jan. 1985), 3–18.

[105] McCarthy, J., Abrahams, P., Edwards, D., Hunt, T., and Levin, M. *LISP 1.5 Programmer's Manual.* MIT Press, 1965.

[106] McDonald, D., Fahlman, S., and Wholey, S. Internal design of CMU Common Lisp on the IBM RT PC. Tech. Rep. CMU-CS-87-157, CMU CS Dept., Sep. 1987.

[107] Meehan, J. *The New UCI LISP Manual.* Lawrence Erlbaum Associates, Hillsdale NJ, 1979.

[108] Miranda, E. BrouHaHa - a portable Smalltalk interpreter. In *Object-Oriented Programming Systems, Languages, and Applications 1987 Conference Proceedings* (Orlando FL, 1987), ACM SIGPLAN, pp. 354–365.

[109] Moon, D. Symbolics architecture. *IEEE Computer 20*, 1 (Jan. 1987), 43–52.

[110] Moon, D. A. Garbage collection in a large Lisp system. In *Proc. 1984 ACM Symposium on LISP and Functional Programming* (Austin TX, Aug. 1984), ACM SIGPLAN/SIGACT/SIGART, pp. 235–246.

[111] Moore II, J. S. The Interlisp virtual machine specification. Tech. Rep. CSL-76-5, Xerox Palo Alto Research Center, Sep. 1976.

[112] Morris Jr., J. B., and Singleton, D. J. The University of Texas 6400/6600 LISP 1.5: an adaptation of MIT LISP 1.5. Tech. Rep. TPB-76, Computation Center, The University of Texas at Austin, Apr. 1967.

[113] Mostow, J. Toward better models of the design process. *AI Magazine 6*, 1 (Spring 1985), 44–57.

[114] Muchnick, S., and Jones, N. *Program Flow Analysis.* Prentice-Hall, Inc., Englewood Cliffs NJ, 1981.

[115] Nordstrom, M. LISP 1.5 interpreter written in Fortran. *SIGPLAN Notices 6*, 5 (July 1971), 6.

[116] Nori, K., Kumar, S., and Kumar, M. Retrospection of the PQCC compiler structure. In *7th Conference on the Foundations of Software Technology and Theoretical Computer Science* (Dec. 1987), K. Nori, Ed., Springer-Verlag, pp. 500–527.

[117] Norman, E. Implementation notes for Univac 1108 LISP. 1968. Description of internal structure.

[118] of Electrical, I., and Engineers, E. Binary floating-point arithmetic. Standard 754-1985, ANSI/IEEE, 1985.

[119] Padget, J., et al. Desiderata for the standardisation of Lisp. In *Proc. 1986 ACM Conference on Lisp and Functional Programming* (Cambridge MA, Aug. 1986), ACM SIGPLAN/SIGACT/SIGART, pp. 54–66.

[120] Pemberton, S., and Daniels, M. *Pascal Implementation: the P4 Compiler.* Ellis Horwood, 1982.

[121] Pereira, F. *C-Prolog User's Manual.* 1984.

[122] Peyton Jones, S. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[123] Pleban, U. Compiler prototyping using formal semantics. In *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction* (Montreal, Canada, 1984), ACM SIGPLAN, pp. 94–105.

[124] Quam, L., and Diffie, W. Stanford Lisp 1.6 Manual. Operating Note 28.7, Stanford Artificial Intelligence Laboratory, 1969.

[125] Quintus Computer Systems. *Quintus Prolog User's Manual.* 1987.

[126] Rees, J., and Clinger, W. Revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices 21*, 12 (Dec. 1986), 37–79.

[127] Rees, J. A., and Adams, N. I. T: a dialect of Lisp or, LAMBDA: the ultimate software tool. In *Proc. 1982 ACM Symposium on LISP and Functional Programming* (Pittsburgh PA, Aug. 1982), ACM SIGPLAN/SIGACT/SIGART, pp. 114–122.

[128] Robison, A. D. The Illinois functional programming interpreter. In *Proceedings of the SIGPLAN 87 Symposium on Interpreters and Interpretive Techniques* (June 1987), pp. 64–73.

[129] Rosenberg, A. Storage mappings for extendible arrays. In *Current Trends in Programming Methodology*, Prentice-Hall, 1978, pp. 263–311.

[130] Rosenschein, S. J., and Katz, S. M. Selection of representations for data structures. In *Proc. Symp. on Artifical Intelligence and Programming Languages* (Rochester NY, Aug. 1977), ACM SIGART/SIGPLAN, pp. 147–154.

[131] Rowe, L., and Tonge, F. Automating the selection of implementation structures. *IEEE Transactions on Software Engineering SE-4*, 11 (Nov. 1978), 494–506.

[132] Russell, S. Compleat guide to mrs. Report KSL-85-12, Computer Science Department, Stanford University, June 1985.

[133] Sammet, J. *Programming Languages: History and Fundamentals*. Prentice-Hall, 1969.

[134] Schwartz, J. T., Dewar, R. B. K., Dubinsky, E., and Schonberg, E. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, 1986.

[135] Scott, D. Data types as lattices. *SIAM Journal on Computing 5* (1976), 522–587.

[136] Shaw, M. Research directions in abstract data structures. In *Proceedings of Conference on Data: Abstraction, Definition, and Structure* (March 1976), pp. 66–68.

[137] Shaw, R. Empirical analysis of a Lisp system. Tech. Rep. CSL-TR-88-351, Computer Systems Laboratory, Stanford University, Feb. 1988.

[138] Shebs, S. T. A Common Lisp implementation of 3-Lisp. 1985. Description of a simple portable implementation.

[139] Shebs, S. T. IDRIL: an interrupt-driven functional language. Technical Report TAMUDCS-82-101-R, Texas A & M University, Oct. 1982.

[140] Shebs, S. T. An machine-description-driven assembler for UCL. 1988. Description of a portable assembler.

[141] Shebs, S. T., and Kessler, R. R. Automatic design and implementation of language datatypes. In *Proceedings of the SIGPLAN 87 Symposium on Interpreters and Interpretive Techniques* (June 1987), pp. 26–37.

[142] Sherman, M. *Paragon: A Language Using Type Hierarchies for the Specification, Implementation, and Selection of Abstract Data Types*. Vol. 189 of *Lecture Notes in Computer Science*, Springer-Verlag, 1985.

[143] Sipala, P. Compact storage of binary trees. *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 345–361.

[144] Smith, B. Reflection and semantics in a procedural language. Tech. Rep. TR-272, MIT Laboratory for Computer Science, 1982.

[145] Free Software Foundation. *GNU Emacs Manual*. 1985.

[146] Steele Jr., G. L. *Common Lisp: the Language*. Digital Press, Burlington MA, 1984.

[147] Steele Jr., G. L. Data representations in PDP-10 MACLISP. AI Memo 420, MIT AI Lab, Cambridge MA, September 1977.

[148] Steele Jr., G. L. An overview of Common Lisp. In *Proc. 1982 ACM Symposium on LISP and Functional Programming* (Pittsburgh PA, Aug. 1982), ACM SIGPLAN/SIGACT/SIGART, pp. 98–107.

[149] Steele Jr., G. L., and Sussman, G. J. Design of a LISP-based microprocessor. *Commun. ACM 23*, 11 (Nov. 1980), 628–645.

[150] Steenkiste, P. LISP on a reduced-instruction-set processor: characterization and optimization. Tech. Rep. CSL-TR-87-324, Computer Systems Laboratory, Stanford University, March 1987.

[151] Steury, C., Leavitt, K., and Likes, K. Portable bignums for PCLS. 1987. Implementation details for class project.

[152] Stoy, J. *Denotational Semantics.* MIT Press, 1977.

[153] Stoyan, H. Early LISP history (1956-1959). In *Proc. 1984 ACM Symposium on LISP and Functional Programming* (Austin TX, Aug. 1984), ACM SIGPLAN/SIGACT/SIGART, pp. 299–310.

[154] Sun Microsystems, Inc. *The* SPARC^{TM} *Architecture Manual.* 1987.

[155] Takeuchi, I., and Okuno, H. A list processor LIPQ. *Review Electronic Communications Laboratory (Japan) 26*, 5-6 (May-June 1978), 767–779.

[156] Teitelman, W., et al. Interlisp reference manual. Tech. Rep., Xerox Palo Alto Research Center, 1974.

[157] Thompson, B., and Thompson, B. Using Pascal to implement functional LISP. *AI Expert 2*, 4 (Apr. 1987), 21–28.

[158] van Katwijk, J. Addressing types and objects in Ada. *Software—Practice and Experience 17*, 5 (May 1987), 319–343.

[159] Wada Laboratory, University of Tokyo. *UtiLisp Manual.* 1988.

[160] Waite, W., and Goos, G. *Compiler Construction.* Springer-Verlag, 1984.

[161] Wand, M. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 496–517.

[162] Warren, D. H. D. Implementing PROLOG - compiling logic programs. DAI Research Report 39,40, University of Edinburgh, 1977.

[163] Weizenbaum, J. Review: LISP 2. *IEEE Transactions on Electronic Computers EC-16*, 2 (Apr. 1967), 236–238.

[164] White, J. LISP/370: a short technical description of the implementation. *ACM SIGSAM Bulletin* (1978).

[165] White, J. L. Address/memory management for a gigantic lisp environment. In *Proc. 1980 LISP Conference* (Stanford CA, 1980), pp. 119–127.

[166] White, J. L. Reconfigurable, retargetable bignums: a case study in efficient, portable Lisp system building. In *Proc. 1986 ACM Conference on Lisp and Functional Programming* (Cambridge MA, Aug. 1986), ACM SIGPLAN/SIGACT/SIGART, pp. 174–191.

[167] Wholey, S., Fahlman, S. F., and Ginder, J. Revised internal design of Spice Lisp. Tech. Rep. S026, CMU CS Dept., Jan. 1985.

[168] Wick, J. Automatic generation of assemblers. Tech. Rep. RR50, Yale University Dept. of Computer Science, December 1975.

[169] Wise, D. Parallel decomposition of matrix inversion using quadtrees. In *1986 International Conference on Parallel Processing* (Aug. 1986), pp. 92–99.

[170] Yokote, Y., and Tokoro, M. The design and implementation of Concurrent-Smalltalk. In *Object-Oriented Programming Systems, Languages, and Applications 1986 Conference Proceedings* (Portland OR, Oct. 1986), ACM SIGPLAN, pp. 331–340.

[171] Yuasa, T. Kyoto Common Lisp documentation. 1985. Notes on internal structure and porting.

[172] Zipf, G. *Human Behavior and the Principle of Least Effort.* Addison-Wesley, 1949.

[173] Zorn, B., Hilfinger, P., Ho, K., and Larus, J. SPUR Lisp: design and implementation. Tech. Rep. UCB/CSD 87/373, Computer Science Division, UC Berkeley, 1987.