J. Allen

The anatomy of LISP

1975

THE ANATOMY OF LISP

© copyright 1975
BY
JOHN ALLEN

to be published by
McGraw-Hill Publishing Co.

This is a review copy. It is not for distribution or duplication.

TABLE OF CONTENTS

CHAPTER PAGE

1 THE ANATOMY OF LISP 1

2 SYMBOLIC EXPRESSIONS 6

- 2.1 Introduction 6
- 2.2 Symbolic Expressions: abstract data structures 9
- 2.3 Trees: representations of Symbolic expressions 12
- 2.4 Primitive Functions 15
- 2.5 Predicates and Conditional Expressions 21
- 2.6 Sequences: abstract data structures 29
- 2.7 Lists: representations of sequences 34
- 2.8 A respite 41
- 2.9 Becoming an expert 44

3 APPLICATIONS OF LISP 53

- 3.1 Introduction 53
- 3.2 Examples of LISP applications 55
- 3.3 Differentiation 56
- 3.4 Data Bases 65
- 3.5 Algebra of polynomials 71
- 3.6 Evaluation of polynomials 75
- 3.7 The great mothers 85
- 3.8 Another Respite 86
- 3.9 Proving properties of programs 88

4 EVALUATION OF LISP EXPRESSIONS 90

- 4.1 Introduction 90
- 4.2 S-expr translation of LISP expressions 95
- 4.3 Symbol tables 97
- 4.4 λ-notation 99
- 4.5 Mechanization of evaluation 103
- 4.6 Examples of *eval* 106
- 4.7 Variables 114
- 4.8 Environments and bindings 117
- 4.9 *label* 122
- 4.10 Functional Arguments and Functional Values 124

4.11	Binding strategies and implementations	135
4.12	Special Forms and Macros	138
4.13	Review	142

5 IMPERATIVE CONSTRUCTS IN LISP 143

5.1	The *prog*-feature	143
5.2	Alternatives to *prog*	151
5.3	Extensions to *eval*	154
5.4	Non-recursive control structures	154
5.5	*eval* with explicit access	156
5.6	*eval* with explicit control	162
5.7	An evaluator for *prog*	167
5.8	Alternatives to *eval*	176
5.9	Function definitions	179
5.10	Rapprochement: In retrospect	182
5.11	The LISP machine	185

6 IMPLEMENTATION OF THE STATIC STRUCTURE OF LISP 189

6.1	Introduction	189
6.2	Representation of S-expressions	190
6.3	Representation of LISP primitives	193
6.4	A few programming techniques	199
6.5	Symbol tables and property-lists	200
6.6	Property-list functions	205
6.7	An *eval* for p-lists	206
6.8	Representation of property-lists	209
6.9	A picture of the atom *NIL*	214
6.10	Input/Output: *read* and *print*	215
6.11	Table searching: Hashing	219
6.12	A first look at *cons*	224
6.13	Storage management: garbage collection	225
6.14	A simple LISP garbage collector written in LISP	227
6.15	A review of the structure of the LISP machine	231
6.16	Implementations of binding	232
6.17	stack implementation of deep binding	234
6.18	stack implementation of shallow binding	235
6.19	Strategies for LISP implementation	241
6.20	Epilogue	242

7 THE DYNAMIC STRUCTURE OF LISP 244

7.1	Introduction	244
7.2	Primitives for LISP	248

7.3 SM: A Simple machine 251
7.4 Implementation of the primitives. 255
7.5 Assemblers 258
7.6 Compilers for subsets of LISP 260
7.7 Compilation of conditional expressions 263
7.8 One-pass Assemblers and fixups. 267
7.9 Compiling and interpreting 271
7.10 A compiler for simple *eval*: the value stack 274
7.11 A compiler for simple *eval* 277
7.12 Efficient compilation 282
7.13 Efficiency: primitive operations 283
7.14 Efficiency: calling sequences 285
7.15 Efficiency: predicates 289
7.16 A compiler for *progs* 291
7.17 Further optimizations. 293
7.18 Functional Arguments 295
7.19 Macros and special forms. 296
7.20 Compilation and variables 297
7.21 Interactive programming 299
7.22 LISP editors 300
7.23 Debugging in LISP 302

review lead into ↓

8 STORAGE STRUCTURES AND EFFICIENCY 305

8.1 Bit-tables 305
8.2 Vectors and arrays 306
8.3 strings and linear LISP 308
8.4 A Compacting collector for LISP. 311
8.5 *rplaca* and *rplacd*. 314
8.6 Applications of *rplaca* and *rplacd* 316
8.7 Numbers 319
8.8 Stacks and Threading 322
8.9 A non-recursive *read* 324
8.10 Dynamic allocation and LISP 328
8.11 Hash techniques . . ?! 330
8.12 Representations of complex data structures 330

9 EPILOG 332

10 PROJECTS 333

10.1 Extensions to *eval*. 333
10.2 Pretty-printing 335
10.3 Data Bases 338
10.4 Syntax-directed processes. 338

10.5 Syntax-directed I/O 343
10.6 Syntax directed computation 347
10.7 One-pass assemblers 348

BIBLIOGRAPHY 349

INDEX 359

TABLE OF CONTENTS

SECTION			PAGE
1	SYMBOLIC EXPRESSIONS		1
	1.1	Introduction	1
	1.2	Symbolic Expressions: abstract data structures	4
	1.3	Trees: representations of Symbolic expressions	7
	1.4	Primitive Functions	10
	1.5	Predicates and Conditional Expressions	16
	1.6	Sequences: abstract data structures	24
	1.7	Lists: representations of sequences	28
	1.8	A respite	35
	1.9	On becoming an expert	39
2	APPLICATIONS OF LISP		48
	2.1	Introduction	48
	2.2	Examples of LISP applications	50
	2.3	Differentiation	51
	2.4	Algebra of polynomials	60
	2.5	Evaluation of polynomials	64
	2.6	The great mothers	75
	2.7	Another Respite	76
	2.8	Proving properties of programs	78
3	EVALUATION OF LISP EXPRESSIONS		80
	3.1	Introduction	80
	3.2	S-expr translation of LISP expressions	85
	3.3	Symbol tables	87
	3.4	λ-notation	89
	3.5	Mechanization of evaluation	93
	3.6	Examples of *eval*	96
	3.7	Variables	104
	3.8	Environments and bindings	106
	3.9	*label*	111
	3.10	Functional Arguments and Functional Values	113
	3.11	Binding strategies.	124
	3.12	special forms	124
	3.13	The *prog*-feature	126

4.12 special forms 122
4.13 The *prog*-feature 123
4.14 Rapprochement: In retrospect 129

5 RUNNING ON THE MACHINE 135

5.1 *evalquote* 136
5.2 A project: extensions to *eval* 138
5.3 A project: Syntax-directed processes 141
5.4 A project: Syntax-directed I/O 145

6 IMPLEMENTATION OF THE STATIC STRUCTURE OF LISP 151

6.1 Introduction 151
6.2 Representation of Symbolic expressions 152
6.3 Representation of LISP primitives 155
6.4 Symbol tables: revisited 159
6.5 A picture of the atom *NIL* 167
6.6 A first look at *cons* 168
6.7 Storage management: garbage collection 169
6.8 Input/Output: *read* and *print* 173
6.9 Symbol table searching: Hashing. 177
6.10 A review of the structure of the LISP machine. 179
6.11 Binding revisited 181
6.12 SM-*eval* 182
6.13 Macros and special forms. 190
6.14 An example of SM-*eval* 195
6.15 Progs in SM-*eval* 196
6.16 An alternative: deep bindings 198
6.17 Epilogue 200

7 THE DYNAMIC STRUCTURE OF LISP 202

7.1 Introduction 202
7.2 SM:A Simple machine 203
7.3 On Compilation 205
7.4 Compilers for subsets of LISP 207
7.5 One-pass Assemblers 215
7.6 A compiler for simple *eval*: the value stack 221
7.7 A compiler for simple *eval*: the control stack. 224
7.8 A compiler for simple *eval* 226
7.9 A project: Efficient compilation 230
7.10 Efficiency: primitive operations 231
7.11 Efficiency: calling sequences 233
7.12 Efficiency: predicates 236

7.13 A compiler for *progs* 237
7.14 Further optimizations. 239
7.15 A project: One-pass assembler 241
7.16 A project: Syntax directed compilation 242
7.17 A deep-binding compiler. 242
7.18 Compilation and global variables 242
7.19 Functional Arguments 244
7.20 Macros and special forms. 244
7.21 Debugging in general. 246
7.22 Debugging in LISP 247

8 STORAGE STRUCTURES AND EFFICIENCY 249

8.1 Bit-tables 249
8.2 Vectors and arrays 250
8.3 strings and linear LISP 252
8.4 *rplaca* and *rplacd*. 257
8.5 Applications of *rplaca* and *rplacd* 260
8.6 Numbers 264
8.7 Stacks and Threading 266
8.8 Dynamic allocation and LISP 271
8.9 Hash linking 273
8.10 Representations of complex data structures 274

9 IMPLICATIONS FOR OTHER LANGUAGES 275

9.1 On LISP and Semantics 275

10 BIBLIOGRAPHY 280

THE ANATOMY OF LISP

> "... it is important not to lose sight of the fact that there is a difference between training and education. If computer science is a fundamental discipline, then university education in this field should emphasize enduring fundamental principles rather than transient current technology."
>
> Peter Wegner, *Three Computer Cultures*

This text is nominally about LISP and data structures. However it in fact covers much broader areas of computer science. The author has long felt that the beginning student of computer science has been getting a distorted and disjointed picture of the field. In some ways this confusion is natural; the field has been growing at such an rapid rate that few of us are prepared to be judged experts in all areas of the discipline. The current alternative seems to be to give a few introductory courses in programming and machine organization followed by relatively specialized courses in more technical areas. The difficulty with this approach is that much of the technical material never gets related. The student's perspective and motivation suffers in the process. This book uses LISP as a means for relating topics which normally get treated in several separate courses. The point is not that we can do this in LISP, but rather that it is natural to do it in LISP. The high-level notation for algorithms is beneficial in explaining and understanding complex algorithms. The use of abstract data structures and abstract LISP programs shows the true intent of structured programming and step-wise refinement. Much of the current work in mathematical theories of computation is based on LISP-like languages. Thus LISP is a formalism for describing algorithms, for writing programs, and for proving properties of algorithms. We use data structures as the main thread in our discussions because a proper appreciation of data structures as abstract objects is a necessary prerequisite to an understanding of modern computer science.

The importance of abstraction obviously goes much further than its appearance in LISP. Abstraction has often been used in other disciplines as a means for controlling complexity. In mathematics, we frequently are able to gain new insights by recasting a particularly intransigent problem in a more general setting. Similarly, the intent of an algorithm expressed in a high-level language like Fortran or PL/1 is more readily apparent than its machine-language equivalent. These are both examples of the use of abstraction. Our use of abstraction will impinge on both the mathematical and the programming aspects. Initially, we will talk about data structures as abstract objects just as the mathematician takes the natural numbers as abstract entities. We will attempt to categorize properties common to data structures and introduce notation for describing functions defined on these abstractions. At this level of discussion we are thinking of our LISP-like language primarily as a notational convenience rather than a computational device. However, after a certain mathematical familiarity has been established it is important to look at our work from the viewpoint of computer science. Here we must think of the computational aspects of our notation.

We must be concerned with the representational problems: implementation on realistic machines, and efficiency of algorithms and data structures. However, it cannot be over-emphasized that our need for understanding is best served at the higher level of abstraction; it is only after a clear understanding of the problem is attained that we should begin thinking about representation. We can exploit the analogy with traditional mathematics a bit further. When we write *sqrt(x)* in Fortran, for example, we are initially only concerned with *sqrt* as a mathematical function defined such that $x = sqrt(x)*sqrt(x)$. We are not interested in the specific algorithm used to approximate the function intended in the notation. Indeed, thought of as a mathematical notation, it doesn't matter how *sqrt* is computed. We might wish to prove some properties of the algorithm which we are encoding. If so, we would only use the mathematical properties of the idealized square root function. Only later, after we had convinced ourselves of the correct encoding of our intention in the Fortran program, would we worry about the computational aspects of the Fortran implementation *sqrt*. Indeed, the typical user will never proceed deeper into the representational mire than this; only if his algorithm is lethargic due to inefficiencies, or inaccurate due to uncooperative approximations, will he look at the actual implementation of *sqrt*.

Thus, just as it is unnecessary to learn machine language to study numerical algorithms, it is also unnecessary to learn machine language to understand non-numerical or data structure processes. The distinction we are making is between data structures and storage structures. That is, data structures are independent of how they are implemented on a machine. Data structures are representations of information chosen to exhibit certain ordering and accessibility relationships between data items. Certainly in the final analysis we cannot ignore storage structures when we are deciding upon the data structures which will encode the algorithm, but the interesting aspects of representation of information can be discussed at the level of data structures with no loss of generality. The mapping of data structures to storage structures is usually quite machine dependent anyway, and consists of bit-pushing, trickery and black magic. We are more interested in ideas than coding tricks. We will see that it is possible, and most beneficial, to structure our programs such that there is a very clean interface between the abstract algorithm and the chosen representation. That is, there will be a set of representation-manipulating programs to test, select or construct elements of the domain; and there will be a program encoding the algorithm. To change representations only requires changes to constructors, selectors and predicates, not to the basic program.

Thus, we will use abstraction as a two-edged sword to control complexity. We will use the notational benefits to express our ideas in a high-level form; we will study the properties of the notation as a computational device for describing an algorithm. One important insight which should be cultivated in this process is the distinction between the concepts of function and algorithm. The idea of function is mathematical and is independent of any notion of computation; the meaning of "algorithm" is computational, the effect of an algorithm being to compute a function. Thus there are typically many algorithms which will compute a specific function.

This text is **not** meant to be a programming manual for LISP. A certain amount of time is spent giving insights into techniques for writing LISP functions. There are two reasons for this. First, the style of LISP programming is quite different from that of "normal" programming. LISP was one of the first languages to exploit the virtues of recursive programming and explore the power of

function-valued variables. Second, and more important, we will spend a great deal of time discussing various levels of implementation of the language. LISP is an excellent medium for introducing standard techniques in data structure manipulation. Techniques for implementation of recursion, implementation of complex data structures, storage management, and symbol table manipulation are easily motivated in the context of language implementation. Many of these standard techniques first arose in the implementation of LISP. But it is pointless to attempt a discussion of implementation unless the reader has a thorough grasp of the language.

Granting the efficacy of our endeavor in abstraction, why study LISP? LISP is at least fifteen years old and many languages now offer themselves with better notation, more efficient running code, or larger varieties of data structure. The difficulty is that the appropriate combination of these features is not present in any other language.

As a programming language, there is only one viable alternative to LISP if we wish to cover this broad scope of topics in a unified approach: invent our own language. Toy languages are suspect for several reasons. The student may suspect (usually for good reason) that he is a subject in a not too clever experiment being performed upon him by his instructor. Having a backlog of fifteen years in experience and example programs should do much to subdue this discomfort. The development of LISP also shows many of the mistakes that the original implementors and designers made. We will point out the flaws and pitfalls awaiting the unwary language designer.

We claim the more interesting aspects of LISP for students of Computer Science lie not in its features as a programming language, but in what it can show about the structure of Computer Science. There is a rapidly expanding body of knowledge unique to Computer Science, neither mathematical nor engineering per se. Much of this area is presented most clearly by studying LISP.

Again there are two ways to look at a high level language: as a mathematical formalism, and as a programming language. LISP is a better formalism than most of its mathematical rivals because there is sufficient organizational complexity present in LISP so as to make its implementation a realistic computer science task and not just an interesting mathematical curiosity. Much of the power of LISP lies in its simplicity. The data structures are rich enough to easily describe sophisticated algorithms but not so rich as to become obfuscatory. Most every aspect of the implementation of LISP and its translators has immediate implications to the implementation of other languages and to the design of programming languages in general.

We will describe language translators (interpreters and compilers) as LISP functions. The structure of these translators when exposed as LISP functions aids immensely in understanding the essential character of such translators. This is partly due to the simplicity of the language, but perhaps more due to our ability to go right to the essential translating algorithm without becoming bogged down in details of syntax.

LISP has very important implications in the field of programming language semantics, and is the dominant language in the closely related study of provability of properties of programs. The idea of proving properties of programs has been around for a very long time. Goldstine and von

Neumann were aware of the practical benefits of such endeavors. J. McCarthy's work in LISP and the Theory of Computation sought to establish formalisms and rules of inference for reasoning about programs. However, the working programmers recognized debugging as the only tool with which to generate a "correct" program, though clearly the non-occurence of bugs is no guarantee of correctness [1]. The sad state of affairs is that the programmer was right. Until very recently techniques for establishing correctness of practical programs simply did not exist.

A recent set of events is beginning to change this.

1. Programs are becoming so large and complex that, even though we write in a high-level language, our intuitions are not sufficient to sustain us when we try to find bugs. We are literally being forced to look beyond debugging.

2. The formalisms are maturing. We know a lot more about how to write "structured programs"; we know how to design languages whose constructs are more amenable to proof techniques. And most importantly, the tools we need for expressing properties of programs are finally being developed.

3. The development of on-line techniques. The on-line system is the only reason that the traditional means of construction and modification of complex programs and systems has been able to survive this long. Sophisticated display editors, debuggers and file handlers have kept us from falling over the edge. The use of these on-line devices in an interactive program constructing system should allow us to actually write correct practical programs.

This enlightened view of programming blends well with the LISP philosophy. We will show that the most natural way to write LISP programs is "structured" in the best sense of the word, being clean in control structure, concise by not attempting to do too much, and independent of a particular data representation.

Many of the existing techniques for establishing correctness originated in McCarthy's investigations of LISP; and some very recent work on mathematical models for programming languages is easily motivated from a discussion of LISP.

Finally there are certain properties of LISP-like languages which make them the natural candidate for interactive program specification. In the chapter on implications of LISP we will characterize "LISP-like" and show how interactive methods can be developed.

This text is primarily designed for undergraduates and therefore an attempt is made to make it self-contained. There are basically five areas in which to partition the topics: the mechanics of the language, the evaluation of expressions in LISP, the static structure of LISP, the dynamic structure of LISP, and the efficient representation of data structures and algorithms. Each area builds on

[1] In truth, it usually meant that no one was using the program.

the previous. Taken as a group these topics introduce much of what is interesting computer science. The first area develops the programming philosophy of LISP: the use of data structures in programming; the language constructs of recursion; and other uncommon control structures. The second area involves a careful study of the meaning of evaluation in LISP gives insights into other languages and to the general question of implementation. The next two areas are involved with implementation. The section on static structure deals with the basic orgainzation of memory for a LISP machine -- be it hardware or simulated in software. The dynamics of LISP discusses the primitive control structures necessary for implementation of the LISP control structures and procedure calls. LISP compilers are discussed here. The final section relates our discussion of LISP and its implementation to the more traditional material of a data structures course. We discuss the problems of efficient representation of data structures. By this point the student should have a better understanding of the uses of data structures and should be motivated to examine these issues.

A large collection of problems has been included. The reader is urged to do as many as possible. The problems are mostly non-trivial; they attempt to be realistic, introducing some new information which the readers should be able to discover themselves. There are also a few rather substantial projects. At least one should be attempted. There is a significant difference between being able to program small problems and being able to handle large projects.

Finally a note on the structure of the text. The emphasis flows from the abstract to the specific, beginning with a precise description of the domain of LISP functions and the operations defined over that domain, and moves to a discussion of the details of efficient implementation of LISP-like languages. The practical-minded programmer might be put-off by the "irrelevant" theory and the theoretical-minded mathematician might be put-off by the "irrelevant" details of implementation. If you lie somewhere between these two extremes, then welcome.

SECTION 1

SYMBOLIC EXPRESSIONS

1.1 Introduction

This book is a study of data structures and programming languages; in particular it is a study of data structures and programming languages centered around the language LISP. However, this is not a manual to help you become a proficient LISP coder. We will study many of the formal and theoretical aspects of languages and data structures as well as examining the practical applications of data structures. We will show that this area of computer science is a discipline of importance and beauty, worthy of careful study. How are we to proceed? How do we introduce rigor into a field whose countenance is as *ad hoc* and diverse as that of programming? We must also bear in mind that the results of our studies are to have practical applications. We must not pursue theory and rigor without proper regard for practice. Our study is not that of pure mathematics; our results will have applications in everyday programming practice. However, for guidance let's look at mathematics. Here is a well-established discipline rich in history and full of results of both practical and theoretical importance. Will our comparison of mathematics and programming languages bear fruit or will it simply show our impudence and naivete? We shall see.

One of the more fertile, yet easily introduced areas of mathematics, is that of elementary number theory. It is easy to introduce because everyone knows something about the natural numbers. Number theory studies properties of a certain class of operations definable over the set **N** of non-negative integers also called natural numbers. A very formal presentation might begin with a construction of **N** from more primitive notions, but it is usually assumed that the reader is familiar with the fundamental properties of **N**. In either case the next step would be to define the class of operations which we would allow on our domain.

We shall begin our study of LISP in a similar manner, as an investigation of a certain class of operations definable over a domain of objects, called Symbolic Expressions. Though most people know something about the natural numbers, we are perhaps not so fortunate when it comes to Symbolic expressions. We must define what we mean by "symbolic expression". Let's look to mathematics again for help. If we asked someone to define the domain **N**, the definition we would receive would depend on how familar that individual was with the properties of the natural numbers.

For most people and purposes, the following characterization of a natural number is satisfactory:

I : A natural number is a sequence of decimal digits.

However this description is mathematically quite superficial and is completely inadequate for the purposes of discussing properties of N. Clearly all of the information we know about the relationships between natural numbers is lacking in this description. The "meaning" of the natural numbers is missing. It is like giving a person an alphabet and rules for forming syntactically correct words but not supplying a dictionary which relates these words to the person's vocabulary.

If pressed for details we might attempt a more adequate characterization like the following:

I I :

1. *zero* is an element of N.
2. If n is in N then the *successor* of n is in N.
3. The only elements of N are those created by finitely many applications of 1. and 2..

Definition I I appears to be completely at the other end of the spectrum; it tells us very little about the appearance of the integers. It gives us an initial element *zero* and a mysterious operation called *successor*, which is supposed to exhibit a new element, given an old one. And unless we are careful about the meaning of *successor*, definition I I will be inadequate. For example if we define the successor of a natural number to be that same number then I I is satisfied but unsatisfactory.

How should we describe *successor* so that our intentions are captured? A sufficient way is to give a definition of *successor* as a mapping or function from natural numbers to natural numbers. To give such an explicit definition requires some notation: let *0* be a notation for *zero*; then we define a function S such that the successor of *zero* -- called *one* -- is denoted by S(*0*), etc.

The characterization of decimal digits given in I is syntactic. The notation itself tells us nothing about the interrelationships between the numbers, but it does give us a notation for representing them. Thus *2* can be used to represent *two* or used as an abbreviation for S(S(*0*)). One benefit of the S-notation is that it explicitly shows the means of construction. That is, it shows more of the properties of the these numbers than just distinguishability. We shall refer to the digit representation as **numerals** and reserve the term, **natural number**, for the abstract object. Thus numerals denote, stand for, or represent the abstract objects called natural numbers; and definition I is better stated as: "a natural number can be represented as a finite sequence of digits".

But notation and syntax are necessary and we must be able to give precise descriptions of syntactic notions. Given a choice between the two previous definitions, I and I I, it appears that I I is more precise. Much less is left to the imagination; given *zero* and a definition of *successor* the definition will act as a recipe for producing elements of N. This style of definition is called an inductive definition. The basic content of an inductive definition of a set of objects consists of three parts:

(1) A description of an initial set of objects; the elements of this set are to be included automatically in the set we are describing in the inductive definition.

IND

(2) Given the description of some existing elements in the set, we are given a means of constructing more elements.

(3) An extremal clause, saying that the only elements in the set are those which gained admittance by either (1) or (2).

Clearly our definition of **N** in terms of *zero* and *successor* is an instance of **IND**: we are defining the set of natural numbers: *zero* is initially included in the set; then applying the second phrase of the definition we can say that *one* is in the set since *one* is the successor of *zero*.

We can recast the positional notation description as an inductive definition.

1. A numeral is a digit

2. if n is a numeral then n followed by a digit is a numeral.

3. The only numerals are those created by finitely many applications of 1. and 2..

In words, "a numeral is a digit, or a numeral followed by a digit".

In this application of **IND**, the initial set has more than one element; namely the ten decimal digits. Also we assume that the questioner knows what "digit" means. This is a characteristic of all definitions: we must stop somewhere in our explication. Notice too that we assume that "followed by" means juxtaposition.

Inductive definitions have been the province of mathematics for many years; however, computer science has encroached a bit and has developed a style of syntax specification called BNF (Backus-Naur Form) equations which has the same intent as that of inductive definitions. Here is the previous inductive definition of "numeral" as a set of BNF equations:

<numeral> ::= <digit>
<numeral> ::= <numeral><digit>

As an abbreviation, the two BNF equations may also be written:

<numeral> ::= <digit>|<numeral><digit>.

A comparison between the BNF and the inductive descriptions of "numeral" should clarify much of the notation, but to be complete we give a more detailed analysis. The symbol "::=" is to be read "is a", the symbol "|" is to be read "or". The strings beginning with "<" and ending with ">" stem from the references to "numeral" and "digit" in 1 and 2; by convention, components of BNF equations which describe elements are enclosed in "<" and ">"; and elements which are given **explicitly** are written without the "< >" fence. Thus "<digit>" is not a numeral but is a description; to make the definition of <numeral> complete we should include an equation like:

<digit> ::= *0* |*1* |*2* |*3* |*4* |*5* |*6* |*7* |*8* |*9*

Juxtaposition of objects implies concatenation of the syntactic objects. Thus "89" is an instance of "<numeral><digit>".

What should be remembered from the discussion in this section? First we wanted and needed precise ways of describing the elements of our study on data structures. We have seen that inductive definitions are a powerful way of describing sets of objects. We have seen a variant of inductive definitions called Backus-Naur Form equations. We will use BNF equations to describe the syntax of our data structures and our language.

Second, we have begun to see the difference between an abstract object and its representation. This distinction has been well studied in philosophy and mathematics, and we will see that this idea has strong consequences for the field of programming and computer science in general. Representation of abstract objects will play a crucial role in this text.

1.2 Symbolic Expressions: abstract data structures

We now wish to apply the techniques and ideas which we have developed in the previous section. We wish to show that careful definitions and use of abstraction will benefit the study of data structures and LISP. To begin our study we should therefore characterize the domain of LISP data structures in a manner similar to what we did for numbers.

Our objects are called **Symbolic Expressions**. Our domain of Symbolic Expressions is named **<sexpr>**. Symbolic expressions are also known as **S-expressions** or **S-exprs**.

The set of symbolic expressions is defined inductively over a base set named **<atom>**. The set **<atom>** can itself be defined inductively. We give the BNF equations for elements of **<atom>** below, but the essential character of the domain is that it contains two kinds of objects: the **literal atoms** and the signed numerals. The elements of **<atom>** are called **atoms**.

<atom>	::= <literal atom>|<numeral>| -<numeral>
<literal atom>	::= <atom letter>|<literal atom><atom letter>|<literal atom><digit>
<numeral>	::= <digit>|<numeral><digit>
<atom letter>	::= *A* |*B* |*C* ...| *Z* [1]
<digit>	::= *0* |*1* |*2* ... |*9*

Thus a literal atom is a string of uppercase letters and digits, subject to the provision that the first character in the atom be a letter.

For example:

atoms	not atoms
ABC123	*2a*
12	*a*
A4D6	*$$g*
NIL	*ABD.*
T	*(A . B)*

The characteristics of atoms which most interest us are their distinguishability: the atom *ABC* is distinguishable from the atom *AB*. That the string *AB* is a part of the string *ABC* is not germane to our current discussion [2]. Similarly, we will seldom need to exploit numerical relationships underlying the numerals. At best we will use simple counting properties. Thus most of our discussions will deal with non-numeric atoms. Most implementations of LISP do however contain a large arithmetic entourage. Many implementations also give a wider class of literal atoms, allowing some special characters to appear; for most of our discussion the above class is quite sufficient.

The domain of Symbolic expressions, called **<sexpr>** is defined inductively over the domain **<atom>** [3].

1. Any element of **<atom>** is an element of **<sexpr>**.

2. If α_1 and α_2 are elements of **<sexpr>**, then the **dotted-pair** $(\alpha_1.\alpha_2)$ is in **<sexpr>**.

Thus **<sexpr>** includes **<atom>** as a proper subset. The notation we chose for the dotted-pairs is the following:

A dotted-pair consists of a left-parenthesis followed by an S-expr, followed by a period, followed by an S-expr, followed by a right-parenthesis.

---

[1] We use ellipses here as a convienient abbreviation.

[2] However, we will discuss such topics in Section on string processing.

[3] We will not give the extremal clause, but it is assumed to hold.

In the definition of <sexpr> we have introduced α_i as match-variables. Greek letters α and β will be used throughout the text in several contexts to designate pattern matches. For example, if we let α_1 be *12* and let α_2 be *ABC* then $(\alpha_1.\alpha_2)$ is *(12 . ABC)* or if we let *(A .(B . C))* be $(\alpha\ .\ \beta)$ then α is *A* and β is *(B . C)*.

Finally here's a BNF description of the full set of S-expressions.

<sexpr> :: = <atom>|(<sexpr> . <sexpr>)

Notice that if we allow floating point numbers as atoms some care needs to be exercised when writing S-expressions. How should *(A.1.2)* be interpreted? Is it the dotted pair *(A . 1.2)*, or is it just an ill-formed expression? Evaluation of such ambiguous constructs will depend on the implementation; such details do not interest us yet.

Examples:	S-exprs	not S-exprs
	A	*A . B*
	(A . B)	*(A . B . C)*
	(((A.B) .C) . (A.B))	*((A . B)))*

Recall our caveat on numerals and numbers. It also applies here. When we described the domain <sexpr> we picked a specific syntactic representation for its elements. It will be a convenient notation since it makes explicit the construction of the composite S-expr from its components [4], and the notation is also consistent with LISP history.

However there is more to the domain <sexpr> than syntax, just as there is more to **N** than positional notation [5]. What are the essential features of S-expressions? Symbolic expressions are either atomic or they have two components. If we are confronted with a non-atomic S-expression then we want a means of distinguishing between the "first" and the "second" component. The "dot notation" does this for us, but obviously "(", ")", and "." of the dotted-pairs are simply notation or syntax. We could have just as well represented the dotted-pair of *A* and *B* as the set-theoretic ordered pair, <*A*,*B*> or any other notation which preserves the essentials of the domain <sexpr>.

The distinctions between abstract objects and their representation are quite important. As we continue our study of more and more complex data structures the use of an abstract data structure instead of one of its representations can mean the difference between a clear and clean program and a confusing and complicated program. There are similar gains for us when we study algorithms defined over these abstract data structures. The less the algorithm knows about the

---

[4] Just as the "successor" notation shows the construction of the numbers from *0*. This kind of notation will be much more useful in LISP, since our interest in data structures will focus on the construction process and the interrelationships between components of an S-expr.

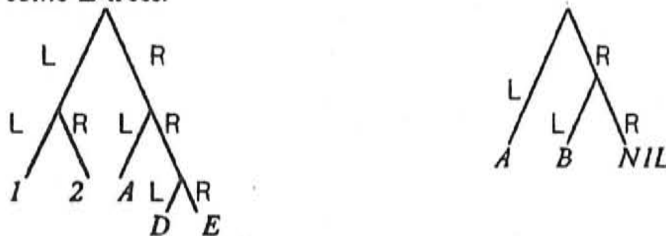
[5] *2*, II in Roman numerals, 10 in binary, "zwei" in German ... are all representations of the same number.

representation of the data structure, the easier it will be to modify or understand that algorithm. Indeed you may have already experienced this phenomenon if you have programmed. A program written in a high-level language is almost always more understandable than its machine-language counterpart -- the benefits of a high-level language are primarily notational rather than computational. The high-level program is more abstract whereas the machine-language program knows a great deal about representations. Finally, if you still doubt that representations make a difference in clarity, try doing long division in Roman numerals. We will say much more about abstraction and representation in algorithms and data structures as we proceed.

1.3 Trees: representations of Symbolic expressions

Besides the more conventional typographical notations, S-expressions also have interesting **graphical** representations. S-exprs have a natural interpretation as a structure which we call a LISP-tree or L-tree.

Here are some L-trees:



We can give an inductive definition:

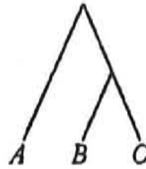
1. Any labelled terminal node is a L-tree. A labelled terminal node is a symbol, •, with an element of <atom> as a subscript. For example $\bullet_{ABC}$.

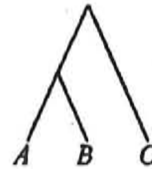
2. If n_1 and n_2 are L-trees then attaching the tails of the arrows, $\rightarrow_L$ and $\rightarrow_R$ to • and the heads of the arrows to n_1 and n_2 also forms an L-tree.

Most important: there are no intersecting branches. We will talk about more general structures later (called list-structures or directed graphs).

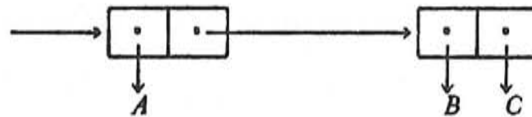
You can see how to interpret S-exprs as L-trees. The atoms are interpreted as terminal nodes; and since non-atomic S-exprs always have two sub-expressions we can write the first subexpression as the left branch of an L-tree and the second sub- expression as the right branch. Typically we, leave off the $_L$ (left) and $_R$ (right) subscripts since it is clear from context which they are. For example:

(A . B) *(A . (B . C))* *((A . B) . C)*





Other representations of LISP-trees which will be more suggestive when we talk about implementation of LISP are:



or equivalently:

This representation (for the S-expr *(A .(B . C))*) is called **box-notation.**

Again please keep in mind the distinction between the abstract object referred to as a symbolic expression and the several representations which we have shown.

The question of representation is so important and will occur so frequently that we introduce notation for a representational mapping, $\mathfrak{R}$. To represent domain D in domain E, we will define a function $\mathfrak{R}_{D \to E}$ which usually will be inductively given, and will express the desired mapping.

For example a representational mapping $\mathfrak{R}_{\text{<sexpr>} \to \text{L-tree}}$ can be given:

$$\mathfrak{R}[\![\text{<atom>}]\!] = \bullet_{\text{<atom>}}$$

and for α and β in <sexpr>:

$\mathfrak{R}[\![(\alpha . \beta)]\!] =$ (tree with branches to) $\mathfrak{R}[\![\alpha]\!]$ $\mathfrak{R}[\![\beta]\!]$

Typically context will determine the appropriate subscript on the $\mathfrak{R}$-mapping; thus we will omit it.

Problems

I Which of the following are dotted-pairs?

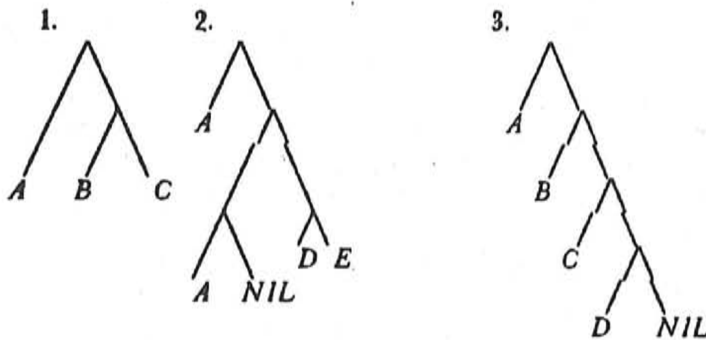
1. *(X . Y)* 2. *((A .(B . C))* 3. *A2* 4. *(X . Y2 . Z)*

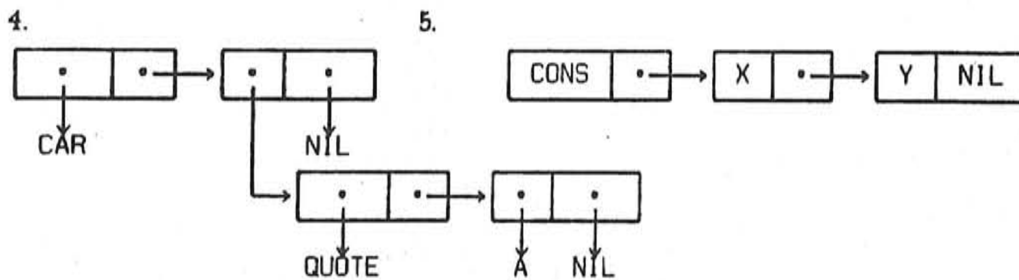
II Write the following as LISP trees:

1. *((A . B).(B . (C . D)))* 2. *(((A . B).C).E)*

3. *((X . NIL).(Y .(Z . NIL)))* 4. *(NIL . NIL)*

III Write the following LISP trees as S-exprs:





1.4 Primitive Functions

So far we have described the domain of abstract objects called Symbolic Expressions and have exhibited several representations for these objects. We will now describe some functions or operations to be performed on this domain. We need to be a bit careful here. We are about to see one of the main differences between mathematics and computer science: mathematics emphasizes the idea of function; computer science emphasizes the idea of algorithm, process, or procedure.

What is a function? Mathematically a function is simply a mapping such that for any given argument in the domain of the function there exists a unique corresponding value. In elementary set theory, a definition of function f involves saying that f is a set of ordered pairs $f = \{ \langle x_1, y_1\rangle, ...\}$; the x_i's are all distinct and the value of the function f for an argument x_i is defined to be the corresponding y_i. With either definition no rule of computation is given to help locate values; with the first definition it is implicit that the internal structure of the mapping doesn't matter; in the set-theoretic definition, the correspondence is explicitly given.

An algorithm or procedure is a process for computing values for a function. The factorial function, $n!$, can be computed by many different algorithms; but thought of as a function it is a set

$$\{\langle 0,1\rangle, \langle 1,1\rangle, \langle 2,2\rangle, \langle 3,6\rangle, ...\langle n,n!\rangle, ...\}$$

For the initial rash of primitives, the distinction between function and algorithm will not be too apparent. However we will soon remedy that situation.

Now for some terminology: the **domain** of a function is the set of all values for which the function is defined; the **range** of a function is the set of all values which the function takes on. A careful definition of a function, f, requires specification of a further set; call it the **domain of discourse**. The domain of discourse, named D, consists of all possible values which may occur as the argument to a function. If the domain of a particular function f coincides with D then f is said to be a **total function** (over D); if there are elements of D which are not in the domain of f then f is a partial function, and f is said to be undefined for those values. For example, the factorial function is typically considered to be partial over the integers, total for the natural numbers, and undefined for negative integers. Thus the concept of "total" or "partial" is relative to a specified domain of discourse. However, a function f total over a domain D_1 can be extended to be total over a domain $D_1 \cup D_2$ by assigning values to $f(d)$ for $d \in D_2$. Thus factorial can be extended to be total over the integers by defining $n!$ to be 0 for n less than 0. We may also extend the range of a function when we extend the domain; thus $f(d)$ need not be in the range of the original f. For example, we added 0 to the range when we extended the factorial function. When we extend the range we must specify what additions are made.

A substantive decision needs to be made on how we are to handle partial functions. Since we are attempting to be reasonably realistic about our modelling of computation we should be as precise as possible in our formalism. We could introduce a class of error values and include them in the range of f; these values would be given as the result of applying f to an argument not in its

domain; or we could simply say that the result is "unspecified" [6]. We shall pick an intermediate position; we shall introduce one new element, $\perp$, called "unspecified" or "undefined", or "bottom" [7]. We will define all our functions over domains augmented with this element; thus constructs like $f(\perp) = a$ are allowed. Think of $\perp$ as covering all anomalous conditions which could be detected and printed as error messages.

As we define new data structures we will frequently want to extend our functions to larger domains. For most of our purposes, a function f defined on (an augmented domain) D will be extended to a larger domain, $D \cup D_1$, by defining $f(d_1) = f(\perp)$ for $d_1 \in D_1$ [8]. Note that $f(\perp)$ need not be $\perp$; however many of the functions which we will examine are defined such that $f(\ldots, \perp, \ldots) = \perp$; that is f returns the undefined value whenever any of its arguments are undefined. Functions which possess this property are called strict functions.

To apply this discussion of $\perp$ to S-exprs we will define an extended domain S to be:

$$S = <sexpr> \cup \{\perp\}.$$

Then we can talk about functions which are total over S or total over <sexpr>, and we will talk about functions which are partial over <sexpr>. Typically when we ask if a function of symbolic expressions is partial or total without specifying a domain, we are asking the question over the natural, unextended domain, <sexpr>.

The first LISP function we consider is the *cons* function which is used to generate S-exprs from less complicated S-exprs. *cons* is called a constructor-function and is a strict, binary function; it is a total function over the domain <sexpr> [9]. Thus *cons* expects two arguments and gives $\perp$ as value whenever either of its arguments is $\perp$. Whenever *cons* is presented with two elements α and β of <sexpr>, *cons*[α;β] returns a new S-expr (α . β). That is, interpreted as a LISP-tree, *cons*[α;β] has a left branch α and has a right branch β.

For example:

cons[*A*; *B*] = (*A* . *B*)
cons[(*A* . *B*); *C*] = ((*A* . *B*) .*C*)

---

[6] Indeed, how "unspecified" manifests itself on a machine will depend on the implementation. Sometimes error messages are given; sometimes it results in an excursion into the subconscious.

[7] "bottom" is sometimes written $\perp$.

[8] The exception to this extension convention involves the definition of predicates which can tell whether or not an arbitrary element is in a specified domain. These predicates always give true or false when applied to any element other than $\perp$.

[9] What we really mean is that either argument to *cons* can be an arbitrary S-expr.

Expressions like the above, which can be evaluated, are called **forms**. S-exprs are forms since they are the constants of our language: the value of a constant is that constant. Function applications are forms: the value is the result of performing the designated function on the designated arguments.

Notice that we are designating function application in LISP by "function name, followed by a list of arguments delimited by '[' and ']' [10]." The '[...]'-notation is part of the LISP syntax and we will reserve '(...)'-notation for the function application of mathematics. In a few places in our discussions the distinction will be important. Typically the distinctions will occur when we wish to distinguish between the LISP algorithm and the mathematical function computed by that algorithm.

We have two strict, unary selector functions, *car* and *cdr* [11], for traversing LISP-trees. We already know the meaning of "strict"; a unary function expects one argument; and a selector function is a data structure manipulating function which will select a component of a composite data structure. Both *car* and *cdr* are selectors since they will select components of non-atomic elements of <sexpr>. Thus *car* and *cdr* are both partial functions over <sexpr>; they give values in <sexpr> only for non-atomic arguments; they give $\perp$ whenever they are presented with an atomic argument.

When given a non-atomic argument, $(\alpha \,.\, \beta)$, *car* returns as value the first subexpression, α; *cdr* (pronounced could-er) returns as value the second sub-expression β.

For example:

$$car[(A \,.\, B)] = A \qquad car[A] = \perp$$
$$cdr[(A \,.\, B)] = B \qquad cdr[(A \,.(B \,.\, C))] = (B \,.\, C)$$
$$car[((A \,.\, B) \,.\, C)] = (A \,.\, B)$$

As with most mathematical theories, we will allow functional composition. The composition of two unary functions f and g is another function, sometimes denoted by $f \circ g$. The value of an expression, $f \circ g[x]$, is the value of $f[g[x]]$. That is, the value of $f \circ g[x]$ is a z such that y is the value of $g[x]$ and z is the value of $f[y]$. $f \circ g$ may be undefined for several reasons: $g[x]$ may be undefined, or $f[y]$ may be undefined.

[10] The syntax equations for forms are given on page 15.

[11] These names are hold-overs from the original implementation of LISP on an IBM 704. That machine had partial-word instructions to reference the address and decrement parts of a machine location. The *a* of *car* comes from "address", the *d* of *cdr* comes from "decrement". The *c* and *r* come from "contents of" and "register". Thus *car* could be read "contents of address part of register".

Here are some examples of composition:

$$car \circ cdr[(A\ .(B\ .\ C))] = car[cdr[(A\ .(B\ .\ C))]] = car[(B\ .\ C)] = B$$
$$cdr \circ cdr[(A\ .(C\ .\ B))] = cdr[cdr[(A\ .(C\ .\ B))]] = cdr[(C\ .\ B)] = B$$
$$cdr[cdr[A]] = \bot$$
$$car[cdr[(A\ .\ B)]] = \bot$$
$$car[cons[X;A]] = X \qquad cdr[cons[Y;X]] = X\ .$$

Notice that the compositions which give result ⊥ do so by resorting to our characterization of *car* and *cdr* as strict functions which have been extended to S.

The composition of many *car* and *cdr* functions occurs so frequently that an abbreviation has been developed. Given such a composition, we select in left-to-right order, the relevant *a*'s and *d*'s in the *car*'s and *cdr*'s. We sandwich this string of *a*'s and *d*'s between a left-hand *c* and a right-hand *r* and give the composition this name.
For example:

$$cadr[x] <= car[cdr[x]]$$
$$caddr[x] <= car[cdr[cdr[x]]]$$
$$cdar[x] <= cdr[car[x]]$$

These compositions are also called *car-cdr*-chains, and are useful in traversing LISP-trees. The "<="- notation is to be read "is defined to be the function ...". This notation is only a temporary convenience and not part of LISP. Very soon we will study what is involved in giving and using definitions in LISP (Section 3.4). For the moment intuition should suffice.

Notice that with these definitions we have introduced variables (over S-exprs). In the sequel lower-case identifiers [12] will be used freely as variables. So for example *Y* and *CAR* are atoms; *y* and *car* could be used as variables. Be clear on the distinction between LISP variables like *x*, *y* or *foo*, and the match variables [13] like α or β. For α and β ranging over S-expressions, $(\alpha\ .\ \beta)$ is a well formed S-expr. The construction, $(x\ .\ y)$, is not well-formed, but *cons*[*x*;*y*] is correct.

It is useful now, however, to introduce some terminology for talking about the components of a function definition. Let

$$f[x_1; ...; x_n] <= \xi$$

represent a typical definition. The name of the function is *f*; the **body** of the function is the expression ξ. The list of variables appearing after the function name are called the **formal parameters**. A function is **applied** using the common notation of **function application**:

$$f[a_1; ...; a_n]$$

---

[12] See page 15 for the BNF equations for <identifier>.

[13] also called meta-variables

The a_i's are called **actual parameters**; for an application to be well formed, the actual parameters must agree in number with the formal parameters of the definition of f and they are to be associated in a one-for-one order, a_i with x_i. Thus in the expression *car[cdr[(A . B)]]* the actual parameter to the *car* function is *cdr[(A . B)]*, and the actual parameter to *cdr* is *(A . B)*. How the actual parameters are to be treated will be a large part of our study.

Once again we are getting a hint of differences between mathematics and computer science. Mathematically speaking, a composition of functions is simply another function -- i.e., a mapping -- and therefore nothing need be said about how to compute composed functions. From a computational point of view, we want to express evaluation of expressions involving composed functions in terms of the evaluation of subexpressions. This would allow us to describe a complex computation in terms of an appropriate sequence of subsidiary computations. One of the more natural ways to evaluate expressions involving compositions is to evaluate the inner-most expressions first, then work outwards. Assume arguments to multi-argument functions are evaluated in left-to-right order. Thus:

cons[car[(A . B)];cdr[(A . (1 . 2))]] = *cons[A;cdr[(A . (1 . 2))]]*
= *cons[A;(1 . 2)]*
= *(A .(1 . 2))*

Evaluation may seem to be a simple operation but looks can be deceiving; evaluation is a very complex process. The value of an expression may depend on the order in which we do things.

For example consider the evaluation of *foo[car[A]; B]* where *foo[x;y]* <= *y*. If we expect *foo* to be a strict function, then *foo[car[A]; B]* must return ⊥ even though it is reasonable to believe that the value of the computation should be *B* since *foo* does not depend on the value of its first parameter. It appears that if we postponed the evaluation of the arguments until those values were actually needed, then at least this problem would be solved. However, the consequences of defining a function to be strict are severe; they cannot be sidestepped by resorting to different schemes for evaluating arguments. There is an alternative strategy in assigning strictness: we could examine the body of the function; if the function uses all its parameters, then its strict. If the function doesn't depend on one or more paramters, then its non-strict. Thus with this interpretation, *foo* is non-strict. We prefer the initial interpretation, reasoning that, if a function is passed bad information, then we wish to know about it, even if the function does not use that specious result.

Strictness is closely related to evaluation schemes for parameter passing. Here are two common techniques:

CBV Evaluate the arguments to a function; pass those evaluated arguments to the function.

This scheme, called Call By Value, is what we were informally using to evaluate the previous examples.

An alternative evaluation process is Call By Name:

CBN Pass the unevaluated arguments into the body of the function.

Assuming *foo* is defined to be strict, then *foo*[*car*[*A*]; *B*] yields $\perp$ under either CBV or CBN. However if we define *foo* to be non-strict then CBV and CBN will both give value *B*. With CBV, x is bound to $\perp$; while with CBN x is bound to *car*[*A*].

Further relationships between evaluation schemes and strictness will be investigated. On page 19 we discuss non-terminating computations. In Section 3 we will discuss evaluation techniques and will give a precise characterization of the evaluation of LISP expressions. On page 18 we will introduce a non-strict language construct but, until that time, intuitive application of CBV will suffice.

What should be kept in mind from this discussion is that we must be careful when discussing the process of evaluation; the function we are characterizing by computing its values will often depend on our choice of evaluation scheme.

Before introducing a further class of LISP expressions we summarize the syntax of the LISP expressions (or forms) allowed so far:

<form> ::= <constant> | <function-part>[<arg>; ...;<arg>] | <variable>

<constant> ::= <sexpr>

<function-part> ::= <identifier>

<arg> ::= <form>

<variable> ::= <identifier>

<identifier> ::= <letter> | <identifier><letter> | <identifier><digit>

<letter> ::= *a* | *b* | *c* ... | *z*

The use of ellipses in the last equation is an abbreviation we have seen before. The use of ellipses in the first equation is different. It is an abbreviation meaning "zero or more occurrences". Thus the equation means a <form> is a <function> followed by the symbol "[" followed by zero or more <arg>'s followed by the symbol "]". This use of ellipses can always be replaced by a sequence of BNF equations. for example, this instance can be replaced by:

<form> ::= <constant> | <function-part>[<arg-list>] | <function-part>[ ] | <variable>
<arg-list> ::= <arg> | <arglist>;<arg>

To increase lucidity we will frequently violate these syntax equations, allowing function names containing special characters, e.g. *fact**, *fib'* or + ; or writing *x+y* instead of +[*x*;*y*]. No attempt will be made to characterize these violations; occurrences of them should be clear from context.

Notice that the class <form> is a collection of LISP expressions which can be evaluated. A <form> is either:

1. a constant: the value is that constant.

2. a function name followed by zero or more arguments: we've said a bit about evaluation schemes for these constructs.

3. a variable: a variable in LISP will typically have an associated value in some environment.

Again we will wait to Section 3.4 for a precise description.

An important constraint on LISP forms which is not covered by the syntax equations is the requirement that functions are defined as being n-ary for some fixed n. Any n-ary function must have **exactly** n arguments presented to it whenever it is applied. Thus *cons*[*A*], *cons*[*A*;*B*;*C*], and *car*[*A*;*B*] are all ill-formed expressions and therefore denote ⊥.

Problems

I. Discuss *cons*[*car*[*x*];*cdr*[*x*]] = *x*.

II. Discuss *cons*[*car*[α];*cdr*[α]] = α.

1.5 Predicates and Conditional Expressions

We cannot generate a very exciting theory based simply on *car*, *cdr*, and *cons* with functional composition. Before we can write reasonably interesting algorithms we must have some way of performing conditional actions. To do this we first need predicates. A LISP predicate is a function returning a value representing truth or falsity. We will represent the concepts of true and false by $\mathfrak{t}$ and $\mathfrak{f}$ respectively. Since these truth values are distinct from elements of S, we will set up a new domain Tr which will consist of the elements, $\mathfrak{t}$ and $\mathfrak{f}$. As usual the extra element ⊥ is included so that we may talk about partial predicates just as we talked about partial functions on <sexpr>. [14].

[14] A word for the inveterate LISP hacker: our use of $\mathfrak{t}$ and $\mathfrak{f}$ marks our first major break from current LISP folklore. The typical LISP trick is to use the atoms *T* and *NIL* rather than $\mathfrak{t}$ and $\mathfrak{f}$ as truth values. Our heresy will disallow some mixed compositions of LISP functions and predicates. We shall not apologize for that and by the end of this chapter you should see why we have deviated.

LISP has two primitive predicates. The first is a strict unary predicate named *atom*; *atom* is total over <sexpr>, and is a special kind of predicate called a **recognizer** or a **discriminator**. Recognizers are used to determine the type of an instance of a data structure. Thus *atom* will return t if the argument denotes an atom, and will return f if the argument is a non-atomic S-expr.

$$atom[A] = atom[NIL] = \mathfrak{t}$$
$$atom[(A\ .\ B)] = \mathfrak{f}$$
$$atom[car[(A\ .\ B)]] = \mathfrak{t}$$
$$atom[\perp] = \perp$$

What should we do about the value of constructs like: *cons[atom[A]; A]*? *atom[A]* gives t, but t is not an element of S and thus is not appropriate as an argument to *cons*. Using our argument of page 11 we extend the domains of the S-expr primitives to

$$S_1 = S \cup Tr$$

For example:

$$car[s] = car[\perp],\quad cons[s;\ A] = cons[\perp;\ A] \text{ for } s \in Tr$$

Since all those functions were strict with respect to undefined we have:

$$atom[\mathfrak{t}] = \perp$$
$$cons[\perp;\ A] = \perp$$

Notice that we now have two separate domains: S-expressions and truth values. Since we will be writing functions over several domains we will need a general recognizer for each domain to assure that the operations defined on each abstract data structure are properly applied. Thus we introduce the recognizer *issexpr* which will give t on the domain of S-exprs, and will give f for any element other than $\perp$ [15].

$$issexpr[(A\ .\ B)] = issexpr[A] = \mathfrak{t}$$
$$issexpr[\mathfrak{t}] = \mathfrak{f}$$
$$issexpr[\perp] = \perp$$

Another primitive predicate we need is named *eq*. It is a strict binary predicate, partial over the set <sexpr>; it will give a truth value only if its arguments are both atomic. It returns t if the arguments denote the same atom; it returns f if the arguments represent different atoms. *eq* is extended to S_1 to yield $\perp$ if either argument to *eq* denotes an element not in the set <atom>.

$$eq[A;A] = \mathfrak{t} \qquad eq[A;B] = \mathfrak{f}$$
$$eq[(A\ .\ B);\ A] = \perp \qquad eq[(A\ .\ B);(A\ .\ B)] = \perp$$
$$eq[eq[A;B];D] = \perp \qquad eq[\perp;x] = \perp$$
$$eq[car[(A\ .\ B)];car[cdr[(A\ .(B\ .\ C))]]] = \mathfrak{f}$$

For completeness' sake we should define a version of *eq*, say eq_{Tr}, which is defined over Tr and acts like *eq*. For expediency's sake we will simply extend the definition of *eq* to S_1 so that it may compare two elements of Tr.

[15] Recall our footnote on page 11.

$$eq[t;t] = t \qquad eq[f;\bot] = \bot$$
$$eq[f;f] = t \qquad eq[t;f] = f$$
$$eq[A;t] = \bot$$

We need to include a construct in our language to effect a test-and-branch operation. The IF-THEN-ELSE operation in LISP is called the conditional expression. It is written:

$$[p_1 \rightarrow e_1;\ p_2 \rightarrow e_2;\ \dots\ ;\ p_n \ \rightarrow e_n]$$

Each p_i is a predicate and therefore takes on values in the set Tr or gives $\bot$; each e_i is an expression which will give a value in S_1. We will restrict the conditionals such that all the e_i must have values in the same domain or be $\bot$; i.e. all be in <sexpr> or all be in Tr.

The meaning (or semantics) of conditionals is:

> We evaluate the p_i's from left to right, finding the first which returns value t. When we find such a p_i, we evaluate the corresponding e_i. The value of the conditional expression is the value computed by that e_i; if all of the p_i's evaluate to f then the conditional expression gives $\bot$. The conditional expression also gives $\bot$ if we come across a p_i which has value $\bot$ before we hit a p_i with value t.

Examples:

$$[atom\ [A] \rightarrow B;\ eq\ [A;(A\ .\ B)] \rightarrow C] = B$$
$$[eq\ [A;(A\ .\ B)] \rightarrow C;\ atom\ [A] \rightarrow B] = \bot$$
$$[atom\ [(A\ .\ B)] \rightarrow B;\ eq\ [A\ ;\ B] \rightarrow C;\ eq\ [car[(A\ .\ B)];\ cdr[(B\ .\ A)]] \rightarrow E] = E$$
$$[eq\ [A;\ A] \rightarrow t;\ atom\ [A] \rightarrow f] = t$$
$$[eq\ [A;\ A] \rightarrow t;\ atom\ [A] \rightarrow B] = \bot$$

Notice that the p_2 expression of the first example is undefined, but the conditional gives value *B* since p_1 gives value t. Thus conditional expressions are non-strict. Note however that non-strictness is relative to a single domain; thus the last example above gives $\bot$ since it contains e_i's of differing domains.

Frequently it is convenient to use a special form of the conditional expression where the final p_n is guaranteed to be true. You can think of lots of predicates which are always true (for example, $eq[1;1]$). A natural predicate is the constant predicate, t; the value of t is always t.

Thus the special form:

$$[p_1 \ \rightarrow e_1;\ \dots;\ t \rightarrow e_n]$$

Now if we know that the previous p_i's are either true or false [16], the final $p_n \rightarrow e_n$-case is a catch-all or otherwise-case which will be executed if none of the previous p_i's give t. Thus the use of t in this context can be read "otherwise"; and the conditional can be read:

"If p_1 is true then e_1, else if p_2 is true then ..., otherwise e_n."

---

[16] and we know that all the e_i's are elements of the same domain.

The introduction of conditional expressions has further widened the gap between traditional mathematical theories and computational theories. Previously we could almost side-step the issue of order of evaluation; it didn't really matter unless $\perp$ got into the act. But now the very definition of meaning of conditionals involves an order of evaluation.

First, the order of evaluation is important from a computational viewpoint on the grounds of efficiency: if we are going to give as value the leftmost e_i whose p_i evaluates to t, then there is no need to compute any of the other e_j's; those values will never be used. A more pressing difficulty is that of partial functions. If we did not impose an order of evaluation on the components of a conditional, then frequently we would attempt to evaluate expressions which would lead to undefined results: $[eq[0;0] \rightarrow 1;\ t \rightarrow car[A]]$ gives 1 using the meaning of conditionals, whereas the expression would be undefined if we were forced to evaluate $car[A]$. This would cause us to lose unnecessarily if we think of an occurrence of $\perp$ during evaluation being mapped to an error message, and causing termination of the computation. But, if we continue to allow $\perp$ as an argument or value to a function, then we can characterize the effect of a conditional expression as a **non-strict** function. Recall, a non-strict function is allowed to return a value other than $\perp$ when one of its arguments is $\perp$; or, put another way, we don't examine the definedness of arguments before applying the function.

For example, let $if(x;y;z)$ be the conditional function [17] computed by: $[x \rightarrow y;\ t \rightarrow z]$. We can define if as a non-strict function such that:

$$if(x;y;z) = \begin{cases} y \text{ if } x \text{ is } t \\ z \text{ if } x \text{ is } f \\ \perp \text{ if } x \text{ is } \perp \end{cases}$$

However there is more to the "strictness" implied by conditional expressions than just making sure that proper arguments are passed on function calls.

Consider the following algorithm:

$$f[x] \Leftarrow [x=0 \rightarrow 1;\ t \rightarrow f[x-1]]$$

Assume the domain of discourse is the positive and negative integers; and assume that f is non-strict. That is, f will try to compute with any (integer) argument it is given. This algorithm defines a function giving 1 for any non-negative integer and is undefined for any other number. From a computational point of view, however, $f[-1]$ appears "undefined" in a different sense from $car[A]$ being "undefined". The computation $f[-1]$ does not terminate and is said to **diverge**. For a partial function like car, we can conceive of giving an error message whenever we attempt to apply the function to an atomic argument. But it stretches one's credulity to expect to include tests like "if the computation $f[a]$ does not terminate then give error No. 15." [18]. From the purely functional point of view, f still defines the partial function which is 1 for the non-negative integers, regardless of how badly it botches things up for negative numbers.

---

[17] Notice we are writing '(...)' rather than '[...]' since we are talking about the function and not the algorithm. See page 12.

[18] Indeed, there are good reasons to be sceptical about the existence of such omniscient tests.

So a computation may be "undefined" for two reasons: it involves a non-terminating computation or it involves applying a partial function to a value not in its domain. Note that the distinction between "undefined" and "diverges" is fuzzy. If we restrict the domain of the above f to the natural numbers, then $f[-1]$ denotes $\perp$ rather than diverges. Or, put another way, "undefined strictness" is a special case of "divergent strictness" where we are able to predict which computations will not terminate. Those cases can be checked by defining the function to be strict over a domain which rules out those anomalies. Thus a case can be made for identifying divergent computations with $\perp$; however there is typically more to non-termination that just "wrong kind of arguments applied".

We want to extend our discussion of strictness to encompass divergence. Recall the discussion on page 14 of

$$foo[x;\ y] \Leftarrow y.$$

Defining *foo* to be strict required that each application of *foo* determine whether either argument denoted $\perp$. If we want *foo* to be strict with respect to divergence, then we must test each argument for divergence. That implies evaluation of the each of the arguments, which in turn implies that if a computation of an argument diverges, then the computation of the function application must also diverge. This implies that it is natural to associate "strict with respect to divergence" with CBV, since in the process of checking for termination, we must compute values. However we already know that if a function is strict then calling style doesn't matter.

In contrast, a non-strict function does not check arguments for divergence, and indeed the divergence of a computation may depend on the calling style. Consider the evaluation of $foo[f[-1];\ B]$ where f was defined above and is total over the integers. This evaluation will diverge under CBV while it converges to B using CBN. We cannot restrict all our functions to be strict if we expect to do any non-trivial computation. That is, we need a function which can determine its value without complete information concerning the values of all of its arguments --a "don't care condition"--. If all the arguments to all subfunctions must be evaluated the computation will not terminate.

The conditional function is such a non-strict function. That is $if(t;q;r)$ has value q without knowing anything about what happens to r. In particular, $if(t;q;\perp) = q$. Likewise $if(f;\perp;r) = r$. Now since if is to be a function and therefore single-valued, if $if(t;q;\perp) = q$ then for any argument x, $if(t;q;x) = q$. Thus $if(t;x;y)$ and $if(f;y;x)$ act like functions of the form:

$$f(x;\perp) = z \text{ implies for every } y \text{ in the domain } f(x;y)=z.$$

Such functions are said to be monotonic functions. Notice that $\perp$ is now carrying an additional "don't-care" interpretation, certainly consistent with its previous assignments when we think of the function being computed by the algorithm.

Even given that a computational definition is desired, there are other plausible interpretations of conditionals. Consider the nonsense definition: $g[x;y] \Leftarrow [lic[x] \rightarrow 1; t \rightarrow 1]$. Clearly, assuming that lic is a total predicate, any value computed by g will be 1. But requiring left-to-right evaluation could spend a great deal of unnecessary computation if lic is a long involved calculation. Questions of evaluation are non-trivial. You should at least be aware that some decisions have been made and others were possible.

What benefits have resulted from our study of $\perp$ and divergence? We should have a clearer understanding of the difference between function and algorithm and a better grasp of the kinds of difficulties which can befall an unwary computation. We have uncovered an important class of detectable errors. The character of these miscreants is that they occur in the context of supplying the wrong **kind** of argument to a function. This kind of error is called a **type fault**, meaning that we expected an argument of a specific type, that is from a specific domain, and since it was not forthcoming, we refuse to perform any kind of calculation. Thus *atom*[f] and *cons*[t;*A*] are undefined since both expect elements of S as arguments. See page for further discussion of type faults. Divergent computations are equally repugnant but there is no general method for testing whether an arbitrary calculation will terminate.

It may not seem like you can do much useful computation with such a limited collection of operations as those proposed in LISP. Things are not quite as trivial as they might seem. In elementary number theory all you have is zero and some simple functions, and elementary number theory is far from elementary. Manipulation of our primitives, with composition, and conditional expressions, coupled with techniques for definition can also become complicated.

Let's apply the LISP constructs which we now have, and define a new LISP function. For example: our predicate *eq* is defined only for atomic arguments. We would like to be able to test for equality of arbitrary S-exprs. What should this more complex equality mean? By equality we mean: as trees, the S-exprs have the same branching structure; and the corresponding terminal nodes are labeled by the same atoms. Thus, we would like to define a predicate, *equal*, such that:

equal [(*A* . *B*);(*A* . *B*)] = t = *equal* [*A*;*A*]
equal [(*A* . *B*);(*B* . *A*)] = f
equal [(*A* . (*B* . *C*));(*A* . (*B* . *C*))] = t
equal [(*A* . (*B* . *C*));((*A* . *B*) . *C*)] = f

Here's an intuitive description of such a predicate named *equal*.

1. If both arguments are atomic then see what *eq* says about them (are they "*eq*"). We can test if they are both atomic by using *atom* and a conditional expression.

2. If one is atomic and the other is not they can't be equal S-exprs.

3. Otherwise both are non-atomic S-exprs. Both have two sub-expressions. Look at both first subexpressions. If the first sub-expressions are not equal then certainly the initial expressions cannot hope to be equal. If, however, the first subexpressions are equal then the question of whether or not the initial expressions are equal depends on the equality (or non-equality) of the second subexpressions. Thus the following definition:

```
equal[x;y] <=     [atom[x] → [atom[y] → eq [x;y];
                               t → f];
                   atom[y] → f;
                   equal [car[x];car[y]] → equal[cdr[x];cdr[y]];
                   t → f]
```

Notice that we use nested conditional expressions in *equal*; e_1 is itself a conditional. Also we have used predicates in the e_i positions at e_3 and e_{11}; this is perfectly reasonable since *equal* is a predicate and thus returns either f or t.

Let's show that *equal* does perform correctly for a specific example. This will also show a complicated evaluation of a conditional expression.

```
equal[(A . B);(A . C)] =     [atom[(A . B)] → [atom[(A . C)] → eq [(A . B);(A . C)];
                                                t → f];
                              atom[(A . C)] → f;
                              equal [car[(A . B)];car[(A . C)]] → equal[cdr[(A . B)];cdr[(A . C)]];
                              t → f]
```

Now using the meaning of conditionals (page 18), we find that p_1 (i.e., *atom[(A . B)]*) and p_2 (*atom[(A . C)]*) when evaluated (in order) give f. We must now evaluate p_3: *equal[car[(A . B)];car[(A . C)]]*. This reduces to *equal[A;A]*, and:

```
equal[A;A] =      [atom[A] → [atom[A] → eq[A;A];
                                t → f];
                   atom[A] → f;
                   equal [car[A];car[A]] → equal[cdr[A];cdr[A]];
                   t → f]
```

This conditional expression will finally evaluate to t. So p_3 in the original call of *equal[(A . B);(A . C)]* is true, and we are faced with the evaluation of e_3 which is *equal[cdr[(A . B);cdr(A . C)]]*. After evaluation of the arguments and evaluation of the conditional expression defining *equal* we will finally return value f. That is, *(A . B)* and *(A . C)* are not equal. Clearly evaluation of LISP expressions in this great detail is not a process which we wish to do very often. It might perhaps be clear that such a process is a likely candidate for execution by a machine.

Notice that throughout this example expressions like *atom[(A . B)]* or *eq[(A . B);(A . C)]* were appearing but were never evaluated because of the way in which we defined evaluation of conditionals.

Finally, to include conditional expressions in our syntax of LISP expressions, we should add:

<form> ::= [<form> → <form>; ...; <form> → <form>] (see page 15)

As usual these syntax equations fail to capture all of our intended meaning. The <form>s appearing in the p_i-position are to be forms taking values in Tr, the truth domain.

Problems

I Evaluate the following:

1. *eq*[*X*;*Y*] 2. *cons*[*X*;*Y*] 3. *car*[(*X* . *Y*)] 4. *car*[*cons*[*X*;*Y*]]

5. *cadr*[(*X* .(*Y* . *NIL*))] 6. *cdar*[(*X* .(*Y* . *NIL*))]

7. *eq*[*cdr*[(*A* . *B*)];*cdr*[(*C* . *B*)]] 8. *atom*[*cons*[(*A* . *B*);(*C* . *D*)]]

9. *cons*[*atom*[*A*];*atom*[(*A* . *B*)]] 10. *eq*[*atom*[*ATOM*];*atom*[*EQ*]]

11. [t → *A*; t → *B*] 12. [f → *A*; t → *B*] 13. [*eq*[*A*;*B*] → *4*]

14. [*atom*[*X*] → *atom*[*X*]; t → *FOO*] 15. [*eq*[*EQ*; *X*] → *A*; *eq*[*A*; *B*] → *B*; t → *C*]

16. *cons*[[*eq*[*A*; *B*] → *1*; t → *FOO*]; *cons*[*A*; *cadr*[(*A* .(*B* .*C*))]]]

17. *equal*[(*A* . *B*);(*A* . *B*)] 18. *eq*[(*A* . *B*);(*A* . *B*)]

II Use the following definition:

```
twist[s] <=    [atom[s] → s;
                t → cons[twist[cdr[s]];twist[car[s]]]]
```

1. Is the function partial or is it total?

Now evaluate:
2. *twist*[*A*] 3. *twist*[(*A* . *B*)] 4. *twist*[((*A* . *B*). *C*)]

III Now try:

```
findem[x;y] <= [atom[x] → [eq[x;y] → T; t → NIL];
                t → cons[findem[car[x];y];findem[cdr[x];y]]]
```

1. Is this function total?

and now evaluate:
2. *findem*[(*A* . *B*);*A*] 3. *findem*[(*B* .(*A* . *C*));*A*] 4. *findem*[(*B* .(*A* . *C*));*C*]
5. *findem*[(*A* . *B*);(*A* . *B*)]

1.6 Sequences: abstract data structures

In several areas of mathematics it is convenient to deal with sequences of information, that is, the problem domain more naturally described as collections of numbers rather than individual numbers. This may either simplify understanding of the problem or simplify the formulation of the functions defined on the domain. Thus several common programming languages include arrays as representations of these mathematical ideas. First we should notice that sequences are data structures. We will have to describe constructors, selectors, and recognizers for them. Also we will explore applications of sequences as data structures suitable for representing many non-trivial problems in computer science.

After a certain familarity is gained in the application of algorithms which manipulate sequences, we will discuss the problems of representation and implementation of this data structure. We will first give an implementation of sequences in terms of S-expressions. That is, we will describe an $\mathfrak{R}$-mapping giving a representation of sequences and their primitive operations in terms of LISP's S-exprs and primitive functions. Later in Section we will discuss low-level implementation of this data structure in terms of conventional machines.

But first we will study sequences as abstract data structures: what are their essential structural characteristics? what properties should be present in a programming language to allow a natural and flexible representation? This discussion will shed light on the important problems of representation and abstraction.

A sequence is an ordered set of elements [19]. For example, (x_1, x_2, x_3), is standard notation for a sequence of the three elements x_1, x_2, and x_3. The length of a sequence is defined to be the number of elements in that sequence. We will allow sequences to have sub-sequences to an arbitrary finite depth. That is, the elements of a sequence will either be individuals or may themselves be sequences. For example, a sequence of length n, each of whose elements are sequences of length m, is a matrix. Here are BNF equations for sequences and their elements:

```
<seq>            ::= ( <seq elem>, ...,<seq elem> ) 20
<seq elem>       ::= <indiv> | <seq>
<indiv>          ::= <literal indiv>|<numeral>| -<numeral>
<literal indiv>  ::= <indiv letter>|<literal indiv><indiv letter>|<literal indiv><digit>
<numeral>        ::= <digit>|<numeral><digit>
<indiv letter>   ::= A |B |C |D |E |F |G |H |I |J |K |L |M |N |O |P |Q |R |S |T |U |V |W |X |Y |Z
<digit>          ::= 0 |1 |2 |3 |4 |5 |6 |7 |8 |9
```

Notice that the structure of <indiv> is the same as that for LISP's <atom>; the only difference is in

[19] For an alternative description of sequences and a discussion of a different view of data structures see page 37.

[20] For the meaning of these ellipses see page 15.

the fonts used for letters and digits. We have made the distinction between LISP atoms and sequence individuals intentionally. Thus (A, (B, C), D, (E, B)) is a sequence of length four, whose second and fourth elements are also sequences. We will use "()" as notation for the empty sequence.

We want to write LISP-like functions operating over sequences, so we will at least need to give constructors, selectors and predicates for sequences [21]. As in the case of Symbolic expressions, we will include the undefined element, and the full domain of sequences will be named

$$Seq = \langle seq \rangle \cup \{\bot\}.$$

As on page 17, we extend the primitive LISP operations to include this new domain, by defining:

$$S_2 = S_1 \cup \langle seq \rangle,$$

and extend each operation appropriately over S_2. Thus, for example:

$$atom[A] = \bot$$
$$car[A] = \bot$$
$$car[(A, B)] = \bot$$
$$cons[A; B] = \bot$$
$$issexpr[(A)] = f$$

We need to define some data structure operations specific to sequences. What are the essential characteristics of a sequence? First, a sequence either is empty or has elements. Thus we will want a predicate to test for emptyness. Next, if the sequence is non-empty, we should be able to select elements. Finally, given some elements, we should be able to build a new sequence from them.

The basic predicate, which tests for emptyness, is called *null*. Predicates on sequences are like predicates on S-expressions, mapping sequences to truth values in Tr [22].

$$null[x] \text{ is } \begin{cases} t \text{ if } x \text{ is the empty sequence, ().} \\ f \text{ if } x \text{ is a non-empty sequence.} \\ \bot \text{ otherwise.} \end{cases}$$

$$null[(\,)] = t$$
$$null[(A, B)] = f$$
$$null[f] = \bot$$

---

[21] Ultimately we will want to give a representation for sequences as S-expressions, and give representations for the sequence operations as operations on S-expressions.

[22] The reason for restructuring LISP predicates should now be apparent to previous users of LISP: if we mapped the truth values to the atoms *T* and *NIL* as is typically done, then we'd have to map truth values of sequence-predicates to representation as sequences. Indeed we would have to perpetuate the fraud for every new abstract data structure domain that we wanted to introduce. Thus we did it right the first time.

Thus *null* gives useable values only for sequences. Since we intend to operate on domains which contain data structures other than sequences, we will need a recognizer to be sure that *null* is not applied to arguments which are not sequences. We will name this recognizer *isseq*.

isseq[(A, B, C)] = t
isseq[A] = f
isseq[*A*] = f *isseq*[t] = f
isseq[()] = t
isseq[⊥] = ⊥

Thus *isseq* is a total predicate over all domains, whereas *null* is only partial, total over <seq>.

While on the subject of predicates, there are a couple more we shall need. The first one is a recognizer, *isindiv*, which will give value t if its argument is an individual, give f if its argument is a sequence, and will give ⊥ otherwise.

The second predicate is the extension of the equality relation to the class of sequence individuals. We shall use the same name, *eq*, as we did for the S-expression predicate. Indeed, whenever we define a new abstract data type we will assume that an appropriate version of *eq* is available for the elements of the base domain. One of our first tasks will be to extend that equality relation to the whole domain. We will do so for sequences later in this section. Equality is a basic relation in mathematics so it is not surprising to see it play an important role here. *eq* is one of the few relations which we shall define across all domains. Functions or predicates like *eq*, which are applicable on several domains, are called polymorphic functions.

Next, the selectors for a (non-empty) sequence include: *first*, *second*, ... ,etc, where:

first[(A, B, C)] = A
second[(A, B, C)] = B
third[(A, B)] = ⊥

It is also convenient to define an "all-but-first" selector, called *rest*.

rest[(A, B, C)] = (B, C)
rest[(B, C)] = (C)
rest[(C)] = ()
rest[C] = ⊥
rest[( )] = ⊥

In conjunction with *rest*, we shall utilize a constructor, *concat*, which is to add a single element to the front of a sequence.

concat[A;(B,C)] = (A, B, C)
concat[A;()] = (A)
concat[(A);(B,C)] = ((A), B, C)
concat[(B,C);A] = ⊥
concat[A; B] = ⊥

The final constructor is called *seq*; it takes an arbitrary number of sequence elements as arguments and returns a sequence consisting of those elements (in the obvious order). Let $\alpha_1, ..., \alpha_n$ be elements of <seq elem>, then:

$$seq[\alpha_1; \alpha_2; ...; \alpha_n] = (\alpha_1, ..., \alpha_n)$$

One question may have come to mind: how do we know when we have a sufficient set of functions for the manipulation of an abstract data structure? How do we know we haven't left some crucial functions out? If we have enough, how do we know that we haven't included too many? Actually, this second case isn't disastrous, but when implementing the functions it would be nice to minimize the number of primitives we have to program. These problems are worthy of study and are the concern of anyone interested in the design of programming languages. We will say a bit more about solutions to these questions beginning on page 33.

Notice that we have been describing the sequence functions without regard to any underlying representation. We have said nothing about these sequence operations except that they construct, test, or select. What we are doing is considering sequences as abstract data structures, suitable for manipulation by LISP-like programs. How sequences are represented on a machine or indeed as S-expressions, is irrelevant. Sequences have certain inherent structural properties and it is those properties which we must understand before we begin thinking about representation on a machine. In the next section we will show how to represent sequences as certain S-expressions and sequence operations as LISP operations on that representation.

First let's develop some expertise in manipulating sequences. The first example will be our promised extension of the equality relation to sequences. We perpetuate the name *equal* from S-exprs, and the basic structure of the definition will parallel that of its namesake; but the components of the definition will involve sequence operations rather than S-expr operations. It might be of value to compare the two predicates. The S-expr version is to be found on page 22.

```
equal[x;y] <=    [null[x] → [null[y] → t;
                              t → f];
                  null[y] → f;
                  equal' [first[x];first[y]] → equal[rest[x];rest[y]];
                  t → f]
```

$$equal''[x;y] \Leftarrow [isindiv[x] \to [isindiv[y] \to eq[x;y]; t \to f]; isindiv[y] \to f; t \to equal[x;y]]$$

Next, we will write a predicate *member* of two arguments *x* and *y*. *x* is to be an individual; *y* is to be a sequence; *member* is to return t just in the case that *x* is an element of the sequence *y*. What does this specification tell us? The predicate is partial. The recursion should be on the structure of *y*; and termination (with value f) should occur if *y* is the empty sequence. If *y* is not empty then it has a first element (call it *z*); compare *z* with *x*. If these elements are identical then *member* should return t; otherwise see if *x* occurs in the remainder of the sequence *y*.

Notes:

1. We cannot use *eq* to check equality since, though *x* is an individual, there is no reason to believe that the elements of *y* need be.

2. Recall that we can get the first element of a sequence with *first*, and the rest of a list with *rest*.

So here's *member*:

$$member[x;y] \Leftarrow [null[y] \to f; equal[first[y];x] \to t; t \to member[x;rest[y]]]$$

Finally here is an arithmetic example to calculate the length of a sequence; that is, the number of elements in the sequence.

$$length[n] \Leftarrow [null[n] \to 0; t \to plus[1;length[rest[n]]]]$$

1.7 Lists: representations of sequences

It is all well and good to be able to write LISP-like functions describing operations on sequences. The algorithms are clean and understandable. However, if we wish to run these programs in a LISP environment, then we had better be prepared to represent both the data structures and the algorithms in terms understandable to LISP [23]. This is the problem of representation. Granted, we could have overcome the problem by explicitly representing sequences directly as LISP S-expressions and could have written functions in LISP which used *car-cdr*-chains to directly manipulate the representations. This misuse of representation is a common fault in LISP

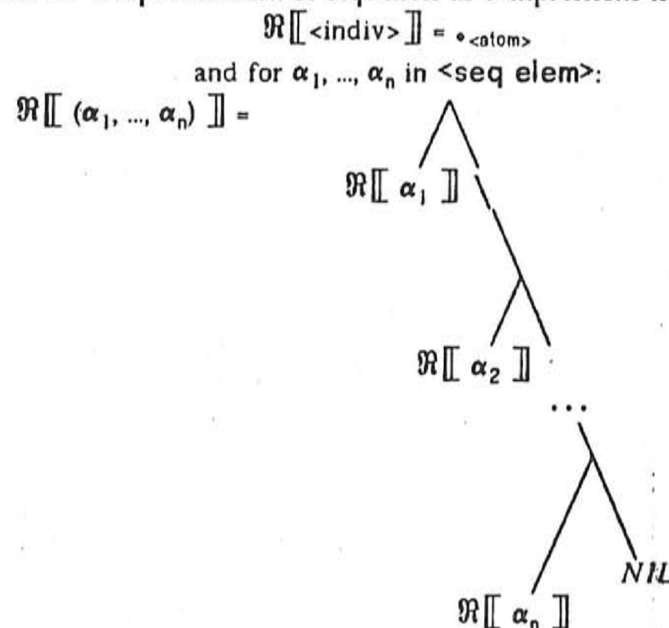
[23] Indeed if we wish LISP to run on a conventional machine we had better be prepared to represent LISP's data structures and algorithms in a manner understandable to conventional hardware. This task is the subject of later chapters in the book.

programming and its practice is to be discouraged. First, the resulting programs are much more difficult to read and debug and understand. More important, the programs are explicitly tied to a specific representation of the abstract data structure; if at some later date it is desired to change the representation, then many programs will have to be rewritten. In Section 2.3 we develop a complex algorithm for differentiation on a class of polynomials, moving from an unclear and highly representation-dependent formulation, to a clear, concise, representation-independent algorithm.

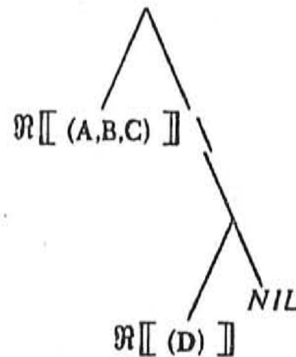
Obviously we will always have to supply a representational bridge between the abstract data structures and algorithms, and their concrete counterparts. One aspect of this study of data structures is to understand what is required to build this bridge and how best to represent these requirements in a programming language.

The first decision to be made is how to represent the abstract data structure; how should we represent sequences as S-expressions? Indeed how should we choose representations in general? Usually there is not just one "best" representation. Some obvious considerations involve the difficulty of implementing the primitive operations (constructors, selectors, recognizers, and predicates) on the abstract data structure. Also we must keep in mind the kinds of algorithms which we wish to write; computation takes time, and since this is computer science we should give consideration to efficiency.

A reasonable choice for a representation of sequences as S-expressions is the following:



That is, the right-hand branch in this LISP-tree representation of a sequence will always point to the rest of the sequence or will be the atom *NIL*. Notice that the description of the ℜ-mapping is recursive. Thus for example:



which will finally expand to *((A .(B . (C . NIL))) .((D . NIL) . NIL))* since ℜ⟦ (A,B,C) ⟧ is *(A .(B .(C .NIL)))* and ℜ⟦ (D) ⟧ is *(D . NIL)*.

You should become fluent in translating between S-expr notation and sequence notation. For convenience sake we will carry over the sequence notation -- **(A, B, C)** -- to that for the representation in LISP -- *(A, B, C)* -- [24] thinking of *(A, B, C)* as an abbreviation for *(A .(B .(C . NIL)))*.

Next, what about a representation for the empty sequence? Looking at the representation of a non-empty sequence it appears natural to take *NIL* as ℜ⟦()⟧ since after you have removed all the elements from the sequence *NIL* is all that is left in the representation. To be consistent then:

ℜ⟦ () ⟧ = *NIL*.

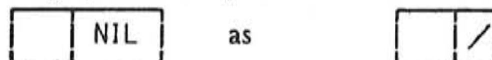
This gives us a complete specification of the ℜ-mapping for the domain; we have represented the abstract domain of sequences in a subset of the domain of Symbolic Expressions. The S-expr representation of a sequence is called a **list**; and we will refer to the abbreviation,

$(\alpha_1, ..., \alpha_n)$ for

$(\alpha_1 .(\alpha_2 . ... (\alpha_n . NIL) ...)$ as **list-notation**.

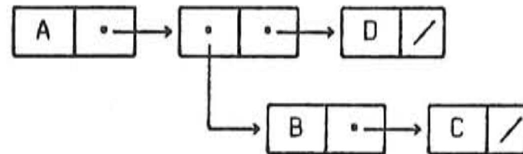
Thus sequences are the abstract data structure; lists are their representation. Since the atom *NIL* takes on special significance in list-notation it is endowed with the special name **list terminator**.

And a notational point: in graphical interpretation of list-notation it is often convenient to write:



[24] Be aware that *A* is an atom and A is a sequence element; they are not the same data structure.

Thus, for example $(A, (B, C), D)$ is:



or, in "dotted-pair" notation: $(A .((B .(C . NIL)).(D . NIL)))$. Finally, in list-notation the commas can be replaced by spaces [25]

e.g. $(A, (B, C), D) = (A\ (B\ C)\ D)$.

but beware: the "dots" in dot-notation are never optional!

that is $(A. (B. C)) \neq (A\ (B\ C))$.

Let's take stock of our position: We have an intuitive understanding of what we mean by "sequence". We have described selectors, constructors, and recognizers, albeit at an abstract level, for manipulating sequences. We have represented our notion of sequences as a subset of the S-expressions called lists. Clearly the final step is to represent our sequence-manipulators as certain LISP functions. Let $first_r$ be a LISP function which will represent the sequence operation *first* [26]. Then for example we might expect:

$$first_r[(A, B, C)] = \mathfrak{R}[\![first[(A, B, C)]]\!] = A.$$

The problem is that this line is not quite right. LISP functions expect their inputs to be S-expressions but (A, B, C) is not an S-expression. To be correct we should have written:

$$first_r[(A .(B . (C . NIL)))] = A$$

It might be argued that (A, B, C) is just a convenient abbreviation for the ugly $(A . (B . (C . NIL)))$, but even so, if we wish the machine to understand and use the abbreviation we must examine the implications of the notation. Clearly it is more reasonable to read and write in list notation. As long as we perform only list-operations on lists there is no reason to look at the

[25] This convention is one of the few instances of a "good" bug. The early LISP papers required full use of commas, but due to a programming error in the LISP output routine, lists were printed without commas. It looked so much better that the bug became institutionalized.

[26] Indeed, once the $\mathfrak{R}$-mapping is defined on the domain it is induced on the operations.

underlying dotted-pair representation [27]. However, it must be remembered that list operations are carried out on the machine using the dotted-pair representation. We might carry out the "list-to-dotted-pair" transformations implicitly, but a machine which evaluates LISP expressions will have to have an explict transformation mechanism.

So a convenient and even necessary part of our representation of sequences is the specification of transformations between the abstract data structure notation and the notation of the underlying representation.

Next we can give representations for the sequence operations. To be precise we should continue our convention of writing the subscript $_r$ on the LISP representation of a sequence operation; thus *seq* is represented by seq_r. In most circumstances no confusion is likely, so we will usually omit the subscript.

In our representation, the construction of a sequence from an arbitrary number of elements will be represented by a LISP function seq_r. We will use *list* interchangeably with seq_r.

$$\mathfrak{R}[\![\ seq\ ]\!] = list$$

$list[\alpha_1;\ \alpha_2;\ ...\ ;\alpha_n]$ generates a list consisting of the α_i arguments. That is, *list* is the appropriately nested composition of *cons*es:

$$cons[\alpha_1;cons[\alpha_2;\ ...\ cons[\alpha_n;NIL]]\ ...]\ .$$

Examples:

$$list[A;B] = (A\ B)$$
$$list[A;B;C] = (A\ B\ C)$$
$$list[cons[A;B];car[(A\ .\ B)]] = ((A\ .\ B)\ A)$$
$$list[A;list[B;C]] = list[A;(B\ C)] = (A\ (B\ C))$$
$$list[] = ()$$
$$list[NIL] = (NIL)$$

Notice that *list* is not strictly a function in the LISP sense; it does evaluate its arguments, but it can take an arbitrary number of them. See page 16. For the moment, *list* is simply a notational abbreviation for nested applications of *cons*. The representation of the selector functions should be apparent from the graphical representation. We leave it as an exercise for the reader to specify representations for these functions; however, here are a few of the other representations:

[27] Indeed, a strong case can be made for never allowing any operations on lists except list operations! See the discussion of type-faults on page 21 and page .

$$\mathfrak{R}[\![\ isindiv\ ]\!] = atom$$

$$\mathfrak{R}[\![\ isseq\ ]\!] = isstrictlist \text{ where:}$$

isstrictlist[x] <= [*atom*[x] → [*eq*[x; *NIL*] → t; t → f] ;
islistelement[*car*[x]] → *isstrictlist*[*cdr*[x]];
t → f]

islistelement[x] <= [*atom*[x] → t; t → *isstrictlist*[x]]

Some practicioners of LISP use this strict definition of lists, so that elements of a list are either atomic or are lists themselves. In practice it is often convenient to allow elements of a list to be arbitrary S-expressions. This more liberal interpretation of lists is expressed by the following recognizer:

islist[x] <= [*atom*[x] →[*eq*[x;*NIL*]→t;t→f]; t→*islist*[*cdr*[x]]]

Thus *(A, (A . B), C)* is a list of three elements. But beware: (A, (A . B), C) is **not** a sequence.

To summarize the accomplishments of this section, we have in effect added a new data structure to the repertoire of LISP. The addition process included:

1. **The abstract operations.** We give constructors, selectors, and predicates for the recognition of instances of the data structure.

2. **The underlying representation.** We must show how the new data structure can be represented in terms of existing data structures.

3. **Abstract operations as concrete operations.** We must write LISP functions which faithfully mirror the intended meaning of the abstract operations when interpreted in the underlying representation.

4. **The input/output transformations.** We should give conventions for transforming to and from the internal representation.

There is another view of this problem of representability of data structures due to J. Morris. Here we use the notion of **transfer functions** whose purpose is to supply mappings between the abstract structure and its representation. Once the transfer functions are given it is usually easy to supply appropriate implementations of the abstract primitives. We need two transfer functions; a write-function, W, to map the representations into the abstract objects; and a read-function, R, to map the abstract objects to their representations.

For the problem at hand, representing sequences, we want R to map from elements of <seq elem> to <sexpr> (see page 24 and page 5); and we want W to map from <sexpr> to <seq elem>. Before we give such R and W, let's see what they will do for us. We could define $first_r$ such that:

$$first_r[x] = W(car[R(x)])$$

Here we have used "(...)" for function application rather than "[...]" which is reserved for LISP evaluation. What the equation says is that given a sequence x, we can map it to the S-expression representation using R; the result of this map is an S-expr and therefore suitable fare for *car*'s delicate palate; the result of the *car* operation is then mapped back into the set of sequence elements by W. The other operations for manipulating sequences can be described similarly. With this introduction, here are appropriate transfer functions:

```
W(e) <=        [isnil[e] → mknull[];
                atom[e] → mkindiv[e];
                t → concat[W(car[e]);W(cdr[e])]
                ]

R(l) <=        [null[l] → NIL;
                isindiv[l] → atomize[l];
                t → cons[R(first[l]);R(rest[l])]
                ]
```

We have seen all of the functions and predicates involved in R and W except the three *atomize*, *mknull* and *mkindiv*. If you think about the implementation of these three functions you would see that they are the identity functions for all practical purposes. However they are only because of the representations that we happened to pick; they need not be so simple. A more careful inspection would show that *mkindiv* expects as input an atomic S-expression and outputs a sequence individual; *atomize* acts conversely. If the representations of the atomic S-expressions were different from the representations of sequence individuals, then we would have some work to do.

The scheme of transfer functions is a more mathematical approach to that outlined above in 1.-4.. We find 1.-4. preferable since it gives a description more in line with what we should expect to implement in a programming language.

To finally review what has transpired since it is a model of what is to come: we developed a new abstract data structure called sequences; discussed notational conventions for writing sequences; described operations and pertinent control devices for writing algorithms; and finally showed that it was possible to represent sequences in the previously developed domain of S-exprs. Thus if we had a machine which could execute S-expr algorithms we could encapsulate that machine within the $\mathfrak{R}$-mapping such that we could write in sequence-notation and have it translated internally to S-expr form; we could write sequence-algorithms and have them execute correctly using the $\mathfrak{R}$-maps of the sequence primitives; and finally it would produce sequence-output rather than the internal S-expr form. To all intents and purposes our augmented LISP machine understands sequences. Indeed, this is the way most LISP implementations are organized; input may either be in S-expr form or list-notation; internally all data structures are stored as S-exprs; all algorithms operate on the S-expr form; and finally, S-exprs which can be interpreted as lists are output in list-notation.

We will approach the other abstract data structure problems in a similar manner, first developing the data structures independent of their representation, and later showing how to represent this

new domain in terms of some previously understood domain. We will see in Section that much of the mapping from input through output can be specified in a natural style and LISP can automatically generate the necessary input and output programs.

Problems

I Discuss *cons*$[\alpha_1;$*cons*$[\alpha_2;\alpha_3]]$ as opposed to *cons*$[\alpha_1;$*cons*$[\alpha_2;$ *cons*$[\alpha_3;$ *NIL*$]]]$ as a representation for $(\alpha_1, \alpha_2, \alpha_3)$.

Problems involving list-notation

I Translate the following lists into S-expr dotted-pair notation.
1. *(A B C)* 2. *(A)* 3. *((A))* 4. *(A (B (C)))* 5. *(NIL)*.

Now go the other way and translate the following S-exprs into list notation.
6. *((A .(B . NIL)).((C . NIL). NIL))* 7. *(NIL . NIL)*
8. *(CONS .((QUOTE .(A . NIL)). NIL))*

II Evaluate the following:
1. *first[(A B)]* 2. *rest[(A B)]*
3. *concat[A;(B C)]* 4. *concat[A;NIL]*
5. *concat[eq[A;A];(A B C)]* 6. *first[rest[(A B)]]*

1.8 A respite

This section contains some hints and notes on the introductory material of this chapter. First a reiteration of a previous admonition: though most of this material seems quite straightforward, the next chapter will begin to show you that things are not all that trivial. LISP is quite powerful. The preceding material is basic and the sooner it becomes second nature to you the better.

A second admonition: besides learning about the basic constructs of the language, the previous material should begin to convince you of the necessity for precise specification of programming languages. In particular we have seen that the process of evaluation of expressions must be spelled out quite carefully. Different evaluation schemes lead to quite different programs. Since evaluation is the business of programming languages we'd better do all we can to make a precise specification.

And a final warning: a major point of this whole book is to stress the proper respect for

abstraction as a tool for controlling complexity in programming, and as a means of writing implementation (representation) independent programs. As we begin writing more complex algorithms, the power of abstraction will become more apparent, but the lessons we learned in representing sequences contain the essential ideas of abstraction and representation. Do not forget them.

We have now seen two examples of abstract data structures. First, the study of Symbolic Expressions was begun without any regard for their implementation. They were deemed to be objects of sufficient interest in their own right [28]. The basic components of our study were the data structures, themselves; the operations on the data structures (*car, cdr, cons, eq* and *atom*), and finally the **control structures** needed by the algorithms which operated on the data structures. The control structures for our LISP algorithms are the conditional expression and recursion. They are called control structures since they are used to direct the flow of the algorithm as it executes. Indeed these three components, data, operations, and control, are the main ingredients of any programming language. Most languages have a superficially richer class of control devices; "while"-statements and "DO"-loops are examples. Most control structures are explicit language constructs, whereas recursion is typically implicit [29].

As we introduce each new abstract data structure we add new operations tailored to its needs. In adding sequences we introduced *first, rest, null,* .... We should also consider the question of introducing new control structures. Again, with sequences we stayed with recursion, though perhaps a simpler control structure which went down the sequence, selecting elements might be more in keeping with the data structure. There is a natural relationship between data structure and control structure; sometimes we can exploit it to good measure. Looking ahead, the iterator *lit* on page is an example of a control regime applicable to sequences. When we consider abstract data structures in future chapters we will again see the three components: data, operations, and control.

The new feature which we considered in discussing sequences was the problem of representation. We showed how to represent sequences in terms of S-expressions. We will continue this pyramiding of data structures in the future; we will consider our work done as soon as we have a representation of our new data structure in terms of an existing one. Finally the suspense will become too great and we will exhibit a representation of the underlying layer of S-expressions. Even later we will discuss efficient representation of data structures, independent of their possible S-expression representation. Indeed, there are lots of data structures which are not best represented as S-expressions. A further consideration appears because of the representational issue; even though we have represented a particular data structure as a complex S-expression we have no business operating on that representation with S-expression functions. Please don't use *car* and *cdr* on lists even though you know what the representation is. You might have noticed that in our

---

[28] But if there is some nagging doubt about the problems of implementation, relax; we'll see plenty of this kind of thing later.

[29] However some languages do require some kind of declaration to the effect that a procedure is recursive.

representation of lists we can find the nth element in a list by using $cad^{n-1}r$. And you know *cadr* is the second element, and that *cdr* is the *rest* of the list. But please keep it a secret. These representation-dependent coding tricks[30] are dangerous. They are really type faults as discussed on page 21 and page .

While we are discussing some of the more practical implications of our work we should not neglect ⊥. How should the be understood. As things currently stand, the appearance of ⊥ in any application of strict functions will immediately cause the termination of the computation. No information other than the fact that ⊥ did appear results from such an occurrence. Indeed if we thought of the evaluation of ⊥ as resulting in a divergent computation, then no information at all would be forthcoming. In reality a LISP implementation will handle computations which we include within the province of ⊥ as error conditions. The computation might be terminated and an error printed; in an interactive implementation the user might be given an opportunity to correct ther error and the computation continued; and alas, some implementations just continue computation with some arbitrary piece of information produced by an excursion into the subsconscious of LISP. We will have much more to say about the implementation of ⊥ in Section .

Well, what does all this have to do with a course in data structures? We will show quite soon that we can exploit abstraction as a means for giving a clear specification of evaluation of LISP expressions, and the representational techniques we will use will involve applications of abstract data structures. A more tangible benefit perhaps should be an increased awareness of the structure and behavior of programming languages, and hopefully the beginnings of a better style of programming.

Another part of our investigation should be to answer the question "What is a data structure?". As we mentioned at the beginning of Section 1.6 there is a different characterization of sequences which will give a different interpretation of data structures. The standard mathematical definition of a sequence is as a function from the integers to a particular domain. Thus a finite sequence **s** might be given as:

$$s = \{<1, s_1>, <2, s_2>, \ldots <n, s_n>\}$$

To select components of **s**, we use ordinary function application: $s(i) = s_i$. Indeed, if you have programmed in a language which has array constructs, you will recognize "application" as the style of notation used. However this is quite different from what we did in the section on sequences. For example, if (A, B, C) is a sequence, **s**, then in the new interpretation we should write:

$$s = \{<1, A>, <2, B>, <3, C>\}$$

Thus **s(2)** is B, etc. What has happened is that what was previously considered to be a data structure has become a function, and the selector functions on the data structure have now become static indices on the function. Or to make things more transparent:

$$s = \{<first, A>, <second, B>, <third, C>\}$$

---

[30] called "puns" by C. Strachey

Then we would write s(*first*) rather than *first*(s) [31]. This idea can easily be applied to S-exprs and their functions. In graphical terms we are representing the structures such that the arcs of the graph are labeled with the selector indices. With L-trees the labeling was implicit: left-branch was *car*; right-branch was *cdr*. With explicit labels on the branches, the trees become unordered. Several languages implement such unordered trees; they are called *structures* in Algol 68 and EL1, and called *records* in Pascal. Several formalisms exploit this view of data structures; in particular the Vienna Definition Language, which is a direct descendant of LISP, represents its data in such a manner.

What then is a data structure? It depends on how you look at it. For our immediate purposes we will try to remain intuitive and informal. We will try to characterize an abstract data structure as a domain and a collection of associated operations and control structures. The operations and control mechanisms should allow us to describe algorithms in a natural manner but should, if at all possible, remain representation independent.

Now for some more mundane tips and tricks on LISP programming. When evaluating or writing functions or (predicates) always keep in mind any restrictions of the function: is it partial? is it total? defined only for lists? are there restrictions on arguments? When taking *car* or *cdr* is the argument non-atomic?

A few tricks were embedded in the problem sets. Recall problem 8 on page 23. The composition *atom*[*cons*[...]] will always evaluate to $\mathfrak{f}$ [32] since the result of *cons*-ing is always non-atomic. In 10., we used atoms with the same letter strings as predicate names, *ATOM* and *EQ*. *ATOM* and *EQ* are perfectly good atoms, and are not the LISP predicates. 14. shows that predicates are perfectly good expressions to evaluate; e_1 is a predicate. Similarly, 16. shows that some conditional expressions may appear within a functional composition.

Notice that *twist* in II is total whereas *findem* is partial. *findem* is partial since *y* must be atomic. Both functions build new trees: *twistem* reverses left- and right-branches recursively; *findem* builds a tree with the same branching structure as *x*, but the terminal nodes contain *T* at the points where the atom *y* appears in the original tree, and *NIL* otherwise.

Now for a representational trick: on page 35 you should have discovered that the value of:

$$cons[\alpha_1; (\alpha_2, ...\alpha_n)] \text{ is: } (\alpha_1, \alpha_2, ... \alpha_n).$$

Notice that $list[\alpha_1;(\alpha_2, ... \alpha_n)]$ is $(\alpha_1\ (\alpha_2 ... \alpha_n))$. So *cons* will add a new element to the front of an existing list. *list* will create a new list whose elements will be the values of the arguments to *list*.

---

[31] G. Steele reports that PPL (Polymorphic Programming Language) at Harvard lets you do this: *car*[*s*] and *s*[*car*] both work.

[32] If it has a value at all! If the computation of the arguments to the *cons* does not terminate or gives $\perp$ then we obviously won't get $\mathfrak{f}$.

Be clear on the difference between the representation of the empty list, *NIL*, and the list consisting of *NIL*, *(NIL)*; *(NIL)* is an abbreviation for *(NIL . NIL)*, which certainly is not *NIL*. List-notation is an abbreviation and can always be translated back into a S-expr.

And an admonition: though our representation of sequences is such that *first*, *rest* and *concat* are identical to *car*, *cdr*, and *cons* respectively, we should use the names *first*, *rest*, and *concat* to make it clear that we are operating on lists.

1.9 On becoming an expert

We have already traced the development of a few LISP algorithms, and we have given a few hints for the budding LISPer. It is time to reinforce these tentative starts with an intensive study of the techniques for writing good LISP programs. This section will spend a good deal of time showing different styles of definition, giving hints about how to write LISP functions, and increasing your familiarity with LISP. For those of you who are impatiently waiting to see some real applications of this strange programming language, we can only say "be patient". The next chapter will develop several non-trivial algorithms, but what we must do first is improve your skills, even at the risk of worsening your disposition.

First some terminology is appropriate: the style of definition which we have been using is called **definition by recursion**. The basic components of such a definition are:

> 1. A basis case: what to compute as value for the function in one or more particularly simple cases. A basis case is frequently referred to as a termination case.

REC

> 2. A general case: what to compute as value for a function, given the values of one or more previous computations on that function.

You should compare the structure of a REC-definition of a function with that of an IND-definition of a set (see IND on page 3). Applications of REC-definitions are particularly useful in computing values of a function defined over a set which has been defined by an IND-definition. For assume that we have defined a set A using IND then a plausible recipe for computing a function f over A would involve two parts: first, tell how to compute f on the base domain of A, and second, given values for some elements of A say $a_1, ..., a_n$, use IND to generate a new element a; then specify the value of f(a) as a function of the known values of $f(a_1), ..., f(a_n)$. That is exactly the structure of REC.

Here is another attribute of IND-definitions: If we have defined a set **A** using IND, assume we wish to prove that a certain property **P** holds for every element of **A**. We need only show that:

1. **P** holds for every element of the base domain of **A**.

PRF

2. Using the technique we elaborated in defining the function **f** above, if we can show that **P** holds for the new element perhaps relying on proofs of **P** for sub-elements, then we should have a convincing argument that **P** holds over all of **A**.

This proof technique is a generalization of a common technique for proving properties of the integers. In that context it is called mathematical induction.

So we are seeing an interesting parallel between inductive definitions of sets, recursive definitions of functions, and proofs by induction. As we proceed we will exploit various aspects of this interrelationship. However our task at hand is more mundane: to develop facility at applying REC to define functions over the IND-domains of symbolic expressions, **S**, and of sequences, **Seq**.

First let's be reassured that the functions we have constructed so far do indeed satisfy REC. Recall our example of *equal* on page 22. The basis case involves a calculation on members of <**atom**>; there we rely on *eq* to distinguish between distinct atoms. The question of equality for two non-atomic S-exprs was thrown back to the question of equality for their *cars* and *cdrs*. But that too, is the right thing to do since the constructed object is in fact manufactured by *cons*, and *car* and *cdr* of that object get the components.

Similar justification for *length* on page 28 can be given. Here the domain is **Seq**. The base domain is the empty sequence, and *length* is defined to give *0* in that case. The general case in the recursion comes from the IND-definition of a sequence[33]. There, given a sequence s, we made a new sequence by adding a sequence element to the the front of s. Again the computation of *length* parallels this construction, saying that the length of this new sequence is one more than the length of s.

For a more traditional example consider the factorial function, n!.

1. The function is defined for non-negative integers.

2. The value of the function for 0 is 1.

3. Otherwise the value of n! is n times the value of (n-1)!.

---

[33] Note (page 24) that we didn't give an explicit IND-definition, but rather a set of BNF equations. The reader can easily supply the explict definition.

It should now be clear how to write a LISP program for the factorial function:

fact[*n*] <= [*eq*[*n*;*0*] → *1*; t → *times*[*n*;*fact*[*sub1*[*n*]]]] [34]

The implication here is that it is somehow easier to compute (n-1)! than to compute n!. But that too is in accord with our construction of the integers using the successor function.

These examples are typical of LISP's recursive definitions. The body of the definition is a conditional expression; the first few branches involve special cases, called **termination conditions.** Then the remainder of the conditional covers the general case-- what to do if the argument to the function is not one of the special cases.

Notice that *fact* is a partial function, defined only for non-negative integers. When writing or reading LISP definitions pay particular attention to the domain of definition and the range of values produced. The following general hints should also be useful:

1. Is it a function or predicate? This information can be used to double-check uses of the definition. Don't use a predicate where a S-expr-valued function is expected.

2. Are there any restrictions on the argument positions? Similar consistency checking as in 1. can be done with this information.

3. Are the termination conditions compatible with the restrictions on the arguments? If it is a recursion on lists, check for the empty list; if it is a recursion of arbitrary S-exprs, then check for the appearance of an atom.

4. Whenever a function call is made within the definition, are all the restrictions on that function satisfied?

5. Don't try to do too much. Be lazy. There is usually a very simple termination case. If the termination case looks messy, there is probably something wrong with your conception of the program. If the general case looks messy, then write some subfunctions to perform the brunt of the calculation.

Use the above suggestions when writing any subfunction. When you are finished, no function will do very much, but the net effect of all the functions acting in concert is a solution to your problem. That is part of the mystery of recursive programming.

As you most likely have discovered, the real sticky business in LISP programming is writing your own programs. But who says programming is easy? LISP at least makes some of your decisions easy. Its structure is particularly spartan. So far there is only one way to write a non-trivial algorithm in LISP: use recursion. The structure of the program goes like that of an inductive argument. Find the right induction hypothesis and the inductive proof is easy; find the right

---

[34] *times* is assumed to be a LISP function which performs multiplication, and *sub1*[*n*] <= *n-1*.

structure to recur on and recursive programming is easy. It's easier to begin with unary functions; then there's no question about what to recur on. The only decision now is how to terminate the recursion. If the argument is an S-expr we typically terminate on the occurrence of an atom; if the argument is a list then terminate on *()*.

First let's consider a slightly more complicated arithmetical example, the Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, ... This sequence is frequently characterized by the following recurrence relation:

$$f(0) = 0$$
$$f(1) = 1$$
$$f(n) = f(n-1)+f(n-2);$$

The translation to a LISP function is easy:

```
fib[n] <=   [eq[n;0] → 0;
             eq[n;1] → 1;
             t → plus[fib[sub1[n]];fib[sub1[sub1[n]]]]]
```

where *plus* is a representation of the mathematical function, +.

A few points can be made here. First, notice that the intuitive evaluation scheme requires many duplications of computation. For example, computation of *fib[5]* requires the computation of *fib[4]* and *fib[3]*. But within the calculation of *fib[4]* we must again calculate *fib[3]*, etc. It would be nice if we could restructure the definition of the function *fib* to stop this duplication of calculation. Since we do wish to run programs on a machine we should give some attention to efficiency. To those with programming experience, the solution is easy: assign the partial computations to temporary variables. The problem here is that our current subset of LISP doesn't contain assignment. There is, however, a very useful trick which we can use.

We will define another function, called *fib'*, on three variables x, y, and n. The variables, x and y, will be used to carry the partial computations. Consider:

```
fib1[n] <= fib'[n;0;1]

fib'[n;x;y] <=      [eq[n;0] → x;
                     t → fib'[sub1[n];plus[x;y];x]]
```

This example is complicated enough to warrant examination. The initial call, $fib_1[n]$, has the effect of calling *fib'* with x initialized to *0* and with y initialized to *1*. The calls on *fib'* within the body of the definition, say the ith such recursive call, has the effect of saving the ith Fibonacci number in x and the $i\text{-}1^{st}$ in y. For example:

```
fib1[4]     = fib'[4;0;1]
            = fib'[3;1;0]
            = fib'[2;1;1]
            = fib'[1;2;1]
            = fib'[0;3;2]
            = 3
```

This same trick of using auxiliary functions can be applied to the factorial example. When viewed computationally, the resulting definition will be more efficient, though the gain in efficiency is not as apparent as that in the Fibonacci example [35]. Thus:

$$fact_1[n] \Leftarrow fact'[n;1];$$

$$fact'[n;x] \Leftarrow [eq[n;0] \rightarrow x;\ t \rightarrow fact'[sub1[n];times[n;x]]]$$

It is clear in these examples that the functions *fact*, $fact_1$ and *fib*, fib_1 are equivalent. Perhaps we should prove that this is so. We presented the crucial ideas for the proof in the discussion on page 40 concerning **IND**, **REC** and **PRF**. We shall examine the question of proofs of equivalence in Section 2.8.

The trick of auxiliary functions is clearly applicable to LISP functions defined over S-exprs:

$$length[n] \Leftarrow [null[n] \rightarrow 0;\ t \rightarrow add1[length[rest[n]]]]$$ [36]

$$length_1[n] \Leftarrow length'[n;0]$$

$$length'[n;x] \Leftarrow [null[n] \rightarrow x;\ t \rightarrow length'[rest[n];add1[x]]]$$

and it seems apparent that *length*[n] is equivalent to $length_1[n]$.

So far our examples have either been numerical or have been predicates. Predicates only require traversing existing S-exprs; certainly we will want to write algorithms to build new S-exprs. Consider the problem of writing a LISP algorithm to reverse a list *x*. The intuitive computation is quite simple. Take elements off of *x* one at a time and put them onto a new list *y*; as initialization, *y* should be *()* and the process should terminate when *x* is empty. Thus:

x	*y*
(A B C D)	*( )*
(B C D)	*(A)*
(C D)	*(B A)*
(D)	*(C B A)*
( )	*(D C B A)*

Here's a plausible *reverse*; notice that we use a sub-function *rev'* to do the hard work and sneak the initialization in on the second argument to *rev'*.

$$reverse[x] \Leftarrow rev'[x;(\ )]$$

$$rev'[x;y] \Leftarrow [null[x] \rightarrow y;\ t \rightarrow rev'[rest[x];concat[first[x];y]]]$$

[35] The fib_1 example improves efficiency mostly by calculating fewer intermediate results. The gain in the $fact_1$ example is involved with the machinery necessary to actually execute the program: the run-time environment, if you wish. We will discuss this when we talk about implementation of LISP in Section . The whole question of: "what is efficient?" is open to discussion.

[36] $add1[x] \Leftarrow x+1$

The function *reverse* builds a list which is the reverse of its argument. Notice that this definition uses an auxiliary function. Sometimes it is more natural to express algorithms this way. We will see a "direct" definition of the reversing function in a moment.

This *reverse* function builds up the new list in a very straightforward mannner, *concat*-ing the elements onto the second argument of *rev'*. Since *y* was initialized to *( )* we are assured that the resulting construct will be a list.

Construction is usually not quite so straightforward. Suppose we wish to define a LISP function named *append* of two list arguments, *x* and *y*, which is to return a new list which has *x* appended onto the front of *y*. For example:

append[(*A B D*);(*C E*)] = (*A B D C E*)
append[*A*;(*B C*)] is undefined. *A* is not a list.
append[(*A B C*);*NIL*] = *append*[*NIL*;(*A B C*)] = (*A B C*)

So *append* is a partial function. It should be defined by recursion, but recursion on which argument? Well, if either argument is *( )* then the value given by *append* is the other argument. The next simplest case is a one-element list; if exactly one of *x* or *y* is a singleton how does that help us discover the recurrence relation for appending? It doesn't help much if *y* is a singleton; but if *x* is, then *append* could give:

concat[*first*[*x*];*y*] as result.

So recursion on *x* is likely. The definition follows easily now.

append[*x*;*y*] <= [*null*[*x*] → *y*; t → *concat*[*first*[*x*];*append*[*rest*[*x*];*y*]]].

Notice that the construction of the result is a bit more obscure than that involved in *reverse*. The construction has to "wait" until we have seen the end of the list *x*. For example:

```
append[(A B C);(D E F)]   = concat[A;append[(B C);(D E F)]]
                          = concat[A;concat[B;append[(C);(D E F)]]]
                          = concat[A;concat[B;concat[C;append[( );(D E F)]]]]
                          = concat[A;concat[B;concat[C;(D E F)]]]
                          = concat[A;concat[B;(C D E F)]]
                          = concat[A;(B C D E F)]
                          = (A B C D E F)
```

We are assured of constructing a list here because *y* is a list and we are *concat*-ing onto the front of it. LISP functions which are to construct list output by *concat*-ing must *concat* onto the front of an existing list. That list may be either non-empty or the empty list, *( )*. This is why the termination condition on a list-constructing function, such as the following function, *dotem*, returns *( )*.

The arguments to *dotem* are both lists assumed to contain the same number of elements. The value returned is to be a list of dotted pairs; the elements of the pairs are the corresponding elements of the input lists. Thus:

dotem[x;y] <= [*null*[x] → ( );
t → *concat*[*cons*[*first*[x];*first*[y]];*dotem*[*rest*[x];*rest*[y]]]]

The first thing to note is the use of both *concat* and *cons*: *concat* is used to build the final list output; *cons* is used to build the dotted pairs. Now if we had written *dotem* such that it knew about our representation of lists, then both functions would have been *cons*. The definition would not have been quite as clear. Look at a computation as simple as *dotem*[*(A)*;*(B)*]. This will involve

concat[*cons*[*A*;*B*];*dotem*[( );( )]]

Now the evaluation of *dotem*[( );( )] returns our needed *( )*, giving

concat[*cons*[*A*;*B*];( )] = *concat*[*(A . B)*;( )] = *((A . B))*

If the termination condition of *dotem* returned anything other than *( )* then the list-construction would "get off on the wrong foot" and would not generate a true list.

As promised on page 44, here is a "direct" definition of *reverse*.

reverse[x] <= [*null*[x] → ( );
t → *append*[*reverse*[*rest*[x]];*concat*[*first*[x];( )]]]

This reversing function is not as efficient as the previous one. Within the construction of the reversed list the *append* function is called repeatedly. You should evaluate something like *reverse*[*(A B C D)*] to see the gross inefficiency.

It is possible to write a directly recursive reversing function with no auxiliary functions, no functions other than the primitives, and no efficiency. We shall do so simply because it is a good example of the process of discovering the general case of the recursion by careful consideration of examples. Let us call the function *rev*.

Let's worry about the termination conditions later. Consider, for example, *rev*[*(A B C D)*]. This should evaluate to *(D C B A)*. How can we construct this list by recursive calls on *rev*? In the following, assume *x* has value *(A B C D)*. Now note that *(D C B A)* is the value of *concat*[*D*;*(C B A)*]. Then *D* is *first*[*rev*[*rest*[x]]] (it is also *first*[*rev*[x]] but that would not help us). How can we get *(C B A)*? Well:

(C B A) = *rev*[*(A B C)*]
= *rev*[*concat*[*A*;*(B C)*]] (we are going after *rest*[x] again)
but first we can get *A* from *x*.
= *rev*[*concat*[*first*[x];*(B C)*]]
= *rev*[*concat*[*first*[x];*rev*[*(C B)*]]]
= *rev*[*concat*[*first*[x];*rev*[*rest*[*(D C B)*]]]]
= *rev*[*concat*[*first*[x];*rev*[*rest*[*rev*[*rest*[x]]]]]]

Finally

$$rev[x] = concat[first[rev[rest[x]]];rev[concat[first[x];rev[rest[rev[rest[x]]]]]]]$$

The termination conditions are simple. First *rev*[( )] gives ( ). Then notice that the general case which we just constructed has two *concats*. That means the shortest list which it can make is of length two. So lists of length one are handled separately: the reverse of such a list is itself. Thus the complete definition should be:

```
rev[x] <= [    null[x] → ( );
               null[rest[x]] → x;
               t → concat[first[rev[rest[x]]];rev[concat[first[x];rev[rest[rev[rest[x]]]]]]]
               ]
```

Problems

I Use the following definition:

```
match[k;m] <=  [null[k] → NO;
                null[m] → NO;
                eq[first[k];first[m]] → first[k];
                t → match[rest[k];rest[m]]]
```

and evaluate:
1. *match*[(X);(X)] 2. *match*[(A B E);(J O E)] 3. *match*[(F O O); (BAZ)]

II Now write your own functions:

1. *among*[x;y] <= ... : *among* is to be a predicate; *x* is an atom; *y* is a list of atoms. *among* is to return f if *x* is not found as an element of *y*; otherwise, *among* is to return t.

   e.g. *among*[A;(A B C)] = *among*[A;(C D E A)] = t
   among[A1;(A2 B2)] = f.

2. *anywhere*[x;y] <= ... : *anywhere* is a predicate; *x* is an atom; *y* is an arbitrary S-expr or list. *anywhere* is to return t just in the case that *x* appears somewhere in *y*.

   e.g. *anywhere*[A;(A B C)] = *anywhere*[A;((A . B). C)] = t
   anywhere[A;(B C D)] = f.

3. *collectpair*[*z*;*x*;*y*] <= ... : x and y are atoms; z is an S-expression or list, some of whose subexpressions, may begin (*x* ...) or (*y* ...). *collectpair* is to return a dotted pair whose *car*-part is a list of all the occurrences of (*x*...) and whose *cdr*-part is a list of all occurrences of (*y* ...).

   e.g. *collectpair*[((*A 1*)((*B . 2*)(*C A 4*)));*A*;*B*] = (((*A 1*)(*A 4*)).((*B . 2*)))

4. *pred*[*x*] <= ... : x is a positive integer. *pred* is a function, returning the predecessor of its argument. The only arithmetic function you may use is *add1*.

   e.g. *pred*[*3*] = *2*; *pred*[*0*] is undefined;
   pred[*add1*[*x*]] = x for $x \geq 0$.

5. *signum*[*x*] <= ...: x is an integer. *signum* returns *NEGATIVE*, *ZERO*, or *POSITIVE* depending on the sign of x. You may use *add1* and *sub1* but no comparision function other than *eq*.

6. *maxdepth*[*l*] <= ...: l is a list. This function is to find the maximum depth of nesting of any sublist in l. Assume that l is a strict list (see page 33); that is, any sub-element is either atomic or is itself a strict list.

SECTION 2

APPLICATIONS OF LISP

"...All the time I design programs for nonexisting machines and add: 'if we now had a machine comprising the primitives here assumed, then the job is done.'

... In actual practice, of course, this ideal machine will turn out not to exist, so our next task --structurally similar to the original one-- is to program the simulation of the "upper" machine.... But this bunch of programs is written for a machine that in all probability will not exist, so our next job will be to simulate it in terms of programs for a next lower level machine, etc., until finally we have a program that can be executed by our hardware...."

E. W. Dijkstra, *Notes on Structured Programming*

2.1 Introduction

There are at least two ways of interpreting this remark of Dijkstra. At the immediate level we note that anyone who has programmed at a level higher than machine language has experienced the phenomenon. For the natural interpretation of programming in a high-level language is that of writing algorithms for the non-existing high-level machine. Typically, however the changes of representation from machine to machine are all done automatically: from high-level, to assembly language, and finally to hardware instructions.

The more fruitful view of Dijkstra's remark is related to our discussions of abstract data structures and algorithms. In this view we express our algorithms and data structures in terms of abstractions independent of how they may be represented in a machine; indeed we can use the ideas of abstraction regardless of whether the formalism will find a representation on a machine. This use of abstraction is the true sense of the programming style called "structured programming". We will see in this chapter how this programming style is a natural result of writing representation-independent LISP programs.

As we have previously remarked, we will see a close relationship between the structure of an algorithm and the structure of the data. We have seen this already on a small scale: list-algorithms recur "linearly" on *rest* to *( )*; S-expr algorithms recur "left-and-right" on *car* and *cdr* to an atom. Indeed, the instances of control structures appearing in an algorithm typically parallel the style of

inductive definition of the data structure which the algorithm is examining. If a structure is defined as:

$$\mathfrak{D} ::= \mathfrak{D}_1 \mid \mathfrak{D}_2 \mid \mathfrak{D}_3$$

e.g. <seq elem> ::= <indiv> | <seq>

then we can expect to find a conditional expression whose predicate positions are filled by the recognizers for the $\mathfrak{D}_i$'s. If the structure is defined as:

$$\mathfrak{D} ::= \mathfrak{D}_1 \ldots \mathfrak{D}_1$$

e.g. <seq> ::= (<seq elem>, ..., <seq elem>)

that is, a homogeneous sequence of elements, then we will have a "linear" recursion like that experienced in list-algorithms[37]. Finally if the structure is defined with a fixed number of components as:

$$\mathfrak{D} ::= \mathfrak{D}_1\ \mathfrak{D}_2\ \mathfrak{D}_3 \ldots \mathfrak{D}_n$$

e.g. <sexpr> ::= (<sexpr> . <sexpr>)

then we can expect a full recursion like that of S-exprs[38].

Thus a data-structure algorithm tends to "pass off" its work to subfunctions which will operate on the components of the data structure. Thus if a structure of type $\mathfrak{D}$ is made up of components of types $\mathfrak{D}_1$, $\mathfrak{D}_2$, $\mathfrak{D}_3$, and $\mathfrak{D}_4$, then the structure of an algorithm f operating on $\mathfrak{D}$ typically involves calls on subfunctions f_1 through f_4 to handle the subcomputations. Each f_i will in turn break up its $\mathfrak{D}_i$.

Thus the type-structure of the call on f would be:

$$f[\mathfrak{D}] = g[f_1[\mathfrak{D}_1];f_2[\mathfrak{D}_2];f_3[\mathfrak{D}_3];f_4[\mathfrak{D}_4]]$$

This is the essence of level-wise programming: we write f, $f_1, \ldots f_4$ independently of the representation of their data structures. f will run provided that the f_i's are available. As we write the f_i's we will probably invoke computations on components of the corresponding $\mathfrak{D}_i$. Those computations are in turn executed by subfunctions which we have to write. This process of elaboration terminates when all subfunctions are written and all data structures have received concrete representations. In LISP this means the lowest level functions are expressed in terms of LISP primitives and the data structures are represented in terms of S-exprs. Thus at the highest level we tend to think of a data structure as a class of behaviors; we don't care about the internal mechanisms which implement that behavior. At the lowest level, machine-language routines simulate one of many possible representations.

---

[37] Indeed there are other forms of control like iteration or *lit* (page) which are more natural for such data structures.

[38] You should have noticed that we are therefore dealing with essentially "context-free" abstract data structures; i.e., those generated by context-free grammars.

This process of elaboration of abstract algorithm and abstract data structure does not invalidate the top-level definition of f it remains intact. We should note, however, that this style of programming is not a panacea; it is no substitute for clear thinking. It only helps control the complexity of the programming process. With this in mind, here are some programming examples.

2.2 Examples of LISP applications

The next few sections will examine some non-trivial problems involving computations on data structures. We will describe the problem intuitively, pick an initial representation for the problem, write the LISP algorithm, and in some cases "tune" the algorithm by picking "more efficient" data representations.

The examples share other important characteristics:

1. We examine the problem domain and attempt to represent its elements as data structures.

2. We reflect on our (intuitive) algorithm and try to express it as a LISP-like data-structure manipulating function.

3. While performing 1 and 2, we might have to modify some of our decisions. Something assumed to be structure might better be represented as algorithm, or the converse might be the case.

4. When the decisions are made, we evaluate the LISP function on a representation of a problem.

5. We reinterpret the data-structure output as an answer to our problem.

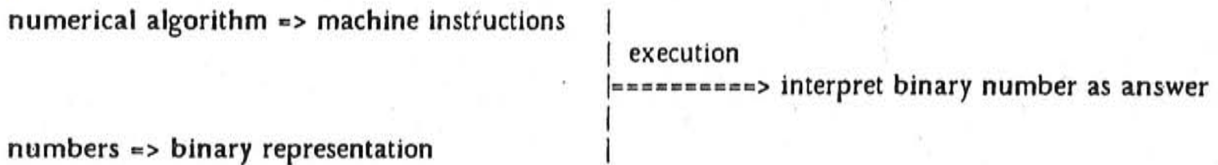
Pictorially in terms of LISP:

```
intuitive algorithm => LISP function    |
                                        | evaluation
                                        |==============> interpret S-expr output as answer
                                        |
problem domain      => S-expressions    |
```

Whenever we write computer programs, whatever language we use, we always go through a similar representation problem. The process is more apparent in a higher-level language like FORTRAN or ALGOL, and is most noticeable in a language like LISP which primarily deals with data structures.

When we deal with numerical algorithms, the representation problem has usually been settled in the transformation from real-world situation to a numerical problem. One has to think more

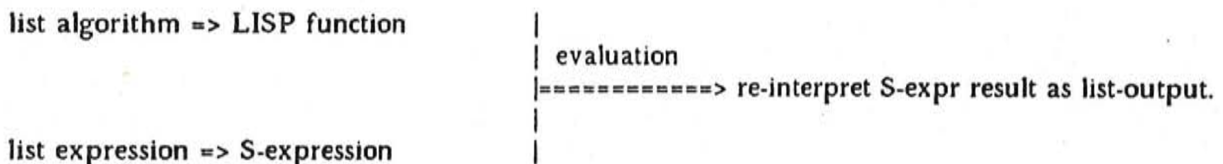
explicitly about representation when we deal with structures like arrays or matrices. We are encoding our information in the array. But the preceding diagram occurs within the machine, even for strictly non-structured numerical calculation.



The encodings are done by the input routines. The result of the execution is presented to the external world by the output routines.

However, it is not until we come to data-structure computations, or non-numerical computations, that the representation problem really becomes undeniable. We have to think more about what we are doing since we lack certain preconceptions or intuitions about such computations. More importantly, however, we are trying to represent actual problems directly as machine problems. We do not attempt to first analyze them into a complex mathematical theory, but should try to express our intuitive theory directly as data-structure computation. This is a different kind of thinking, due wholly to the advent of computers. Indeed the field of computation has expanded so much as to obsolete the term "computer". "Structure processor" is more indicative of the proper level at which we should view "computers".

We have already seen a simple example of the representation problem in the discussion of list-notation beginning in Section 1.6.



The following sections deal with representation of complex data structure problems in LISP.

2.3 Differentiation

This example will describe a rudimentary differentiation routine for polynomials in several variables. We will develop this algorithm through several stages. We will begin by doing a very direct, but representation-dependent, implementation. We will encode polynomials as special LISP lists and will express the differentiation algorithm as a LISP program operating on that

representation. When this program is completely specified we will then scrutinize it, attempting to see just how much of the program and data structure is representation and how much is essential to the expression of the algorithm.

You should recognize two facts about the differentiation algorithm for polynomials: First, the algorithm operates on forms (or expressions) as arguments and returns forms as values. Typically we think of the arguments and values to functions as being simple values, or the results of evaluating complex forms, not forms themselves. Second, and more important, you should realize that the algorithm for differentiation is recursive! The question of differentiation of a sum is thrown back on the differentiation of each summand. Similar relationships hold for products, differences, and powers. As with all good recursive definitions, there must be some termination conditions. What are the termination conditions here? Differentiation of a variable, say x, with respect to x is defined to be 1; differentiating a constant, or a variable not equal to x with respect to x gives a result of zero. This problem should begin to sound like that of the IND-definitions of sets (in this case the set of polynomials) and the associated REC-definitions of algorithms (in this case differentiation of polynomials). If this is the mold into which our current problem fits, then our first order of business is to give an inductive definition of our set of polynomials. Though polynomials can be arbitrarily complex, involving the operations of addition, multiplication, negation, and exponentiation, their general format is very simple if they are described in our LISP-like notation where the operation precedes its operands. If we assume that binary plus, times, and exponentiation are symbolized by +, *, and ↑, we will write +[*x*;2] instead of the usual infix notation *x*+2. The general term for this LISP-like notation is **prefix notation**.

Here are some examples of infix and prefix representations:

infix	prefix
*x***z*+2*y*	+[*[*x*;*z*]; *[2;*y*]]
*x***y***z*	*[*x*;*[*y*;*z*]]

We will now give an inductive definition of the set of polynomials we wish to consider. The definition will involve an inductive definition of terms.

1. terms are polynomials.

2. If p_1 and p_2 are polynomials then the "sum" of p_1 and p_2 is a polynomial.

where:

1. constants and variables are terms

2. If t_1 and t_2 are terms then the "product" of t_1 and t_2 is a term.

3. If t_1 is a variable and t_2 is a constant then "t_1 raised to the t_2th power" is a term.

4. If t_1 is a term the "minus" t_1 is a term.

Armed with prefix notation we can now give a BNF description of the above set:

<poly> ::= <term> | <plus>[<poly>;<poly>]

<term> ::= <constant> | <variable> | <times>[<term>;<term>] | <expt>[<variable>;<constant>]
::= <minus><term>

<constant> ::= <numeral>

<plus> ::= +

<times> ::= *

<expt> ::= ↑

<minus> ::= -

<variable> ::= <identifier>

It is easy to write recursive algorithms in LISP; the only problem is that the domain (and range) of LISP functions is S-exprs, not the polynomials which we need. So we need a way to represent arbitrary polynomials as S-exprs. We will do the representation in lists rather than S-exprs. Let $\Re$ be a function mapping polynomals to their representation such that a variable is mapped to its uppercase counterpart in the vocabulary of LISP atoms. Thus:

$\Re[\![$<variable>$]\!]$ = <literal atom>.

Let constants (numerals), be just the LISP numerals; these are also respectable LISP atoms. Thus:

$\Re[\![$<numeral>$]\!]$ = <numeral>.

Since we have now specified a representation for the base domains of the inductive definition of our polynomials, it is reasonable to think about the termination cases for the recursive definition of differentiation.

We know from differential calculus that if u is a constant or a variable then:

du/dx = 1 if $x = u$
0 otherwise

We will represent the d-operator as a binary LISP function named *diff*. The application, du/dx will be represented as *diff*[*u*;*x*]. Now since constants and variables are both represented as atoms, we can check for both of these cases by using the predicate *atom*. Thus a representation of the termination cases might be:

diff[*u*;*x*] <= [*isindiv*[*u*] → [*eq*[*u*;*x*] → *1*; t → *0*] ....

Notice we write the abbreviation, *isindiv* instead of $isindiv_r$. You should be a bit wary of our definition already: *diff*[*1*;*1*] will evaluate to *1*.

Now that we have covered the termination case, what can be done for the representation of the remaining class of terms and polynomials? That is, how should we represent sums and products?

First, we will represent the operations ∗, +, -, and ↑ as atoms:

$$\Re[\![+]\!] = PLUS$$
$$\Re[\![*]\!] = TIMES$$
$$\Re[\![-]\!] = MINUS$$
$$\Re[\![\uparrow]\!] = EXPT$$

We will now extend the mapping $\Re$ to occurrences of binary operators by mapping to three-element lists:

$$\Re[\![\alpha[\beta_1;\beta_2]]\!] = (\Re[\![\alpha]\!], \Re[\![\beta_1]\!], \Re[\![\beta_2]\!]).$$

Unary applications will result in two-element lists: $\Re[\![\alpha[\beta]]\!] = (\Re[\![\alpha]\!]; \Re[\![\beta]\!])$.

For example:

$$\Re[\![+[x;\ 2]]\!] = (PLUS\ X\ 2)$$

A more complicated example: the polynomial,

$$x^2 + 2yz + u$$

will be translated to the following prefix notation:

$$+[\uparrow[x;2];\ +[*[2;*[y;z]];\ u]]$$ [39]

From this it's easy to get the list form:

(PLUS (EXPT X 2) (PLUS (TIMES 2 (TIMES Y Z)) U))

Now we can complete the differentiation algorithm for + and ∗. We know:

d[*f* + *g*]/dx = d*f*/dx + d*g*/dx.

We would see

$u = \Re[\![f + g]\!] = (PLUS, \Re[\![f]\!], \Re[\![g]\!])$

Where: $second[u] = \Re[\![f]\!]$

$third[u] = \Re[\![g]\!]$. [40]

---

[39] This is messier than it really needs to be because we assume that + and ∗ are binary. You should also notice that our $\Re$-mapping is applicable to a larger class of expressions than just <poly>. Look at $(x + y)*(z + 2)$.

[40] As we intimated earlier, we have done an evil thing here. We have tied the algorithm for symbolic differentiation to a specific representation for polynomials. It is useful to use a specific representation for the moment and repent later. In particular, see page 57, page 63 and page .

The result of differentiating u is to be a new list of three elements:

1. the symbol *PLUS*.
2. the effect of *diff* operating $\Re[\![f]\!]$
3. the effect of *diff* operating $\Re[\![g]\!]$

Thus another part of the algorithm:

eq [*first*[u];*PLUS*] → *list* [*PLUS*; *diff*[*second*[u];x];*diff*[*third*[u];x]].

d[$f*g$]/dx is defined to be $f*$ dg/dx + g *df/dx.

So here's another part of *diff*:

eq[*first*[u];*TIMES*] → *list*[*PLUS*;
list[*TIMES*; *second*[u];*diff* [*third*[u];x]];
list[*TIMES*;*third*[u];*diff* [*second*[u];x]]]

Finally, here's an example. We know:

$$d[x*y + x]/dx = y + 1$$

```
Try:
diff [(PLUS (TIMES X Y) X); X]
         =list [PLUS; diff[(TIMES X Y); X]; diff [X;X]]
         =list [PLUS;
                  list [PLUS;
                           list [TIMES; X; diff [Y;X]];
                           list [TIMES; Y; diff [X;X]]];
                  diff [X;X]]

         =list [PLUS;
                  list [PLUS;
                           list [TIMES; X ;0];
                           list [TIMES; Y;1]];
                  1 ]

         =(PLUS (PLUS (TIMES X 0) (TIMES Y 1)) 1)
```

which can be interpreted as:

$$x*0 + y*1 + 1 .$$

Now it is clear that we have the right answer; it is equally clear that the representation leaves much to be desired. There are obvious simplifications which we would expect to have done before we would consider this output acceptable. This example is a particularly simple case for algebraic simplification. We can easily write a LISP program to perform simplifications like those expected here: like replacing $0*x$ by 0, and $x*1$ by x. But the general problem of writing simplifiers, or indeed of recognizing what is a simplification, is quite difficult. A whole branch of computer science has grown up around symbolic and algebraic manipulation of expressions. One of the crucial parts of such an endeavor is a sophisticated simplifier.

Points to note

This problem of representation is typical of data structure algorithms (regardless of what language you use). That is, once you have decided what the intuitive problem is, pick a representation which makes your algorithms clean (only later should you worry about efficiency). In the next set of examples we will see a series of representations, each becoming more and more "efficient" and each requiring more "knowledge" being built into the algorithm.

Here's the whole algorithm for differentiation using + and *.

```
diff[u;x] <=
        [isindiv[u] → [eq [x;u] → 1; t → 0];
         eq [first [u]; PLUS] → list   [PLUS;
                                         diff [second [u]; x];
                                         diff [third [u]; x]]
         eq [first[u]; TIMES] → list   [PLUS;
                                         list   [TIMES;
                                                 second[u];
                                                 diff [third [u]; x]];
                                         list   [TIMES;
                                                 third [u];
                                                 diff [second[u]; x]]];
         t → ⊥]
```

As we mentioned earlier, the current manifestation of *diff* encodes too much of our particular representation for polynomials. The separation of algorithm from representation is beneficial from at least two standpoints. First, changing representation should have a minimal effect on the structure of the algorithm, but *diff* knows that variables are represented as atoms and a sum is represented as a list whose *first*-part is *PLUS*. Second, readability of the algorithm suffers greatly.

How much of *diff* really needs to know about the representation and how can we improve the readability of *diff*?

To begin with the uses of *first*, *second*, and *third* are not particularly mnemonic [41]. We used *second* to get the first argument to a sum or product and used *third* to get the second. We used *first* to extract the operator.

Let's define the selectors:

$$op[x] <= first[x]$$
$$arg_1[x] <= second[x]$$
$$arg_2[x] <= third[x]$$

Then *diff* becomes:

[41] It must be admitted, however, that they are more readable than *car-cdr*-chains.

```
diff[u;x] <=
        [isindiv[u] → [eq [x;u] → 1; t → 0];
         eq [op[u]; PLUS] → list    [PLUS;
                                     diff [arg1 [u]; x];
                                     diff [arg2 [u]; x]]
         eq [op[u]; TIMES] → list   [PLUS;
                                     list   [TIMES;
                                             arg1 [u];
                                             diff [arg2 [u]; x]];
                                     list   [TIMES;
                                             arg2 [u];
                                             diff [arg1 [u]; x]]];
        t → ⊥]
```

Still, there is much of the representation present. Recognition of variables and other terms can be abstracted from the representation. We need only recognize when a term is a sum, a product, a variable or a constant. To test for the occurrence of a numeral we shall assume a unary LISP predicate called *numberp* which returns t just in the case that its argument is a numeral. Then, in terms of the current representation, we could define such recognizers or predicates as:

```
               issum[x] <= eq[op[x];PLUS]
              isprod[x] <= eq[op[x];TIMES]
             isconst[x] <= numberp[x]
isvar[x] <= [isindiv[x] → not[isconst[x]]; t → f]
          samevar[x;y] <= eq[x;y]
```

Using these predicates we can rewrite *diff* as:

```
diff[u;x] <=
        [isvar[u] → [samevar[x;u] → 1; t → 0];
         isconst[u] → 0;
         issum[u] → list     [PLUS;
                              diff [arg1 [u]; x];
                              diff [arg2 [u]; x]]
         isprod[u] → list    [PLUS;
                              list   [TIMES;
                                      arg1 [u];
                                      diff [arg2 [u]; x]];
                              list   [TIMES;
                                      arg2 [u];
                                      diff [arg1 [u]; x]]];
        t → ⊥]
```

Readability is certainly improving, but the representation is still known to *diff*. When we build the result of the sum or product of derivatives we use knowledge of the representation. Rather it would be better to define:

```
makesum[x;y] <= list[PLUS;x;y]
makeprod[x;y] <= list[TIMES;x;y]
```

Then the new *diff* is:

```
diff[u;x] <=
        [isvar[u] → [samevar[x;u] → 1; t → 0];
         isconst[u] → 0;
         issum[u] → makesum[diff [arg1 [u]; x];diff[arg2 [u]; x]]
         isprod[u] → makesum[   makeprod[arg1 [u];diff[arg2 [u]; x]];
                                 makeprod[arg2 [u];diff [arg1 [u]; x]]];
         t → ⊥]
```

Notice that *diff* is much more readable now and, more importantly, the details of the representation have been relegated to subfunctions. Changing representation simply requires supplying different subfunctions. No changes need be made to *diff*. There has only been a slight decrease in efficiency. The termination condition in the original *diff* is a bit more succinct. Looking back, we first abstracted the selector functions: those which selected components; then we abstracted the recognizers: the predicates telling which term was present; then we modified the constructors: the functions which make new terms. These three components of programming: selectors, recognizers, and constructors, will appear again on page in discussing McCarthy's abstract syntax.

The *diff* algorithm is abstract now, in the sense that the representation of the domain and the representation of the functions and predicates which manipulate that domain have been extracted out. This is our $\mathfrak{R}$-mapping again; we mapped the domain of <poly>'s to lists and mapped the constructors, selectors, and recognizers to list-manipulating functions. Thus the data types of the arguments u and x are <poly> and <var> respectively, not list and atom. To stress this point we should make one more transformation on *diff*. We have frequently said that there is a substantial parallel between a data structure and the algorithms which manipulate it. Paralleling the BNF definition of <poly> on page 53, we write:

```
diff[u;x] <=
        [isterm[u] → diffterm[u;x]
         issum[u] → makesum[diff [arg1 [u]; x];diff[arg2 [u]; x]]
         t → ⊥].

diffterm[u;x] <=
        [isconst[u] → 0;
         isvar[u] → [samevar[x;u] → 1; t → 0];
         isprod[u] → makesum[   makeprod[arg1 [u];diff[arg2 [u]; x]];
                                 makeprod[arg2 [u];diff [arg1 [u]; x]]];
         t → ⊥]
```

To satisfy our complaint of page 53 that *diff*[*1*; *1*] gives a defined result, we should also add:

*diff**[*u*; *x*] <= [*isvar*[*x*] → [*ispoly*[*u*] → *diff*[*u*; *x*]];
t → ⊥]

Finally, notice that our abstraction process has masked the order-dependence of conditional expressions. Exactly one of the recognizers will be satisfied by the form *u*.

Problems

1. Extend the version of *diff* of your choice to handle differentiation of powers such as ↑[*x*; *3*].

2. Extend *diff* to handle unary minus.

3. Extend *diff* to handle differentiation of the trignometric functions, *sin* and *cons* and their composition with polynomials. For example it should handle $sin^2x + cos(x^3 + 5x - 2)$.

4. Write an algorithm to handle integration of polynomials.

2.4 Algebra of polynomials

Assume we want to perform addition and multiplication of polynomials and assume that each polynomial is of the form $p_1 + p_2 + ... + p_n$ where each term, p_i, is a product of variables and constants. The two components of each term are a constant part called the coefficient, and the variable part. We shall assume without loss of generality that the set of variables which appear in the polynomials are lexicographically ordered, e.g. $x < y < z, ...,$ and assume that each variable part obeys that ordering; thus we would insist that xzy^2 be written xy^2z. We do not assume that the terms are ordered within the polynomial; thus $x + xy$ and $xy + x$ are both acceptable. We further assume that the variables of each p_i be distinct and that no p_i have *0* as coefficient. The standard algorithm for addition of $\Sigma^n_{i=1}p_i$ with $\Sigma^m_{j=1}q_j$ says you can combine a q_j with a p_i if the variable parts of these terms are identical. In this case the resulting term has the same variable part but has a coefficient equal to the sum of the coefficients of p_i and q_j. For example if p_i is $2x$ and q_j is $3x$ the sum term is $5x$. We will examine four representations of polynomials, before finally writing any algorithms. To aid in the discussion we will use the polynomial $x^2 - 2y - z$ as our canonical example.

First representation:

We could use the representation of the differentiation example. This would represent our example as:

(PLUS (TIMES 1(EXPT X 2)) (PLUS (TIMES -2 Y) (TIMES -1 Z)))

The above conventions specify an unambiguous representation for our class of polynomials. Strictly speaking we did not need to impose the ordering on the set of variables. However, we need to imposed some additional constraints before we have data structures which are well-suited to the class of polynomial algorithms we wish to represent.

Second representation:

We are really only interested in testing the equality of the variable parts; we will not be manipulating them in any other way. So we might simply represent the variable part as a list of pairs; each pair contains a variable name and the corresponding value of the exponent. Using our knowledge of the forms of polynomials and the class of algorithms we wish to implement, we write Σp_i as:

((rep of p_1), (rep of p_2), ...)

which would make our example look like:

((TIMES 1 ((X . 2))), (TIMES -2 ((Y . 1))), (TIMES -1 ((Z . 1))))

Is this representation sufficient? Does it have the flexibility we need? It does suffice but it is still not terribly efficient. We are ignoring too much of the structure in our class of polynomials.

Third representation:

What do we know? We know that the occurrence of variables is ordered in each variable part; we can assume that we know the class of variables which may appear in any polynomial. So instead of writing x^2y^3z as

((X . 2) (Y . 3) (Z . 1)), we could write:

(2 3 1) assuming x, y, z are the only variables.

In a further simplification, notice that the *TIMES* in the representation is superfluous. We always multiply the coefficient by the variable part. So we could simply *cons* the coefficient onto the front of the variable part representation.
Let's stop for some examples.

term	representation
$2xyz$	*(2 1 1 1)*
$2x^2z$	*(2 2 0 1)*
$4z^3$	*(4 0 0 3)*

Thus our canonical polynomial would now be represented as:

((1 2 0 0) (-2 0 1 0) (-1 0 0 1))

This representation is not too bad; the *first*-part of any term is the coefficient; the *rest*-part is the variable part. So, for example the test for equality of variable parts is simply a call on *equal.*

Now let's start thinking about the structure of the main algorithm.

Fourth representation:

The algorithm for the sum must compare terms; finding like terms it will generate an appropriate new term, otherwise simply copy the terms. When we pick a p_i from the first polynomial we would like to find a corresponding q_j with the minimum amount of searching. This can be accomplished if we can order the terms in the polynomials. A natural ordering can be induced on the terms by ordering the numerical representation of the exponents. For sake of argument, assume that a maximum of two digits will be needed to express the exponent of any one variable. Thus the exponent of x^2 will be represented as *02*, or the exponent of z^{10} will be represented as *10*. Combining this with our ordered representation of variable parts, we arrive at:

term	representation
$43x^2y^3z^4$	*(43, 020304)*
$2x^2z$	*(2, 020001)*
$4z^3$	*(4, 000003)*

Now we can order on the numeric representation of the variable part of the term. One more change of representation, which will result in a simplification in storage requirements:

represent $ax^Ay^Bz^C$ as *(a . ABC)*.

This gives our final representation.

((1 . 20000) (-2 . 100) (-1 . 1))

Note that *20000 > 100 > 1.*

Finally we will write the algorithm. We will assume that the polynomials are initially ordered and will write the algorithm so as to maintain that ordering. Each term is a dotted pair of elements: the coefficient and a representation of the variable part.

As in the previous differentiation example, we should attempt to extract the algorithm from the representation.

We shall define:

coef[x] <= *car*[x] and *expo*[x] <= *cdr*[x].

To test the ordering we will use the LISP predicate:

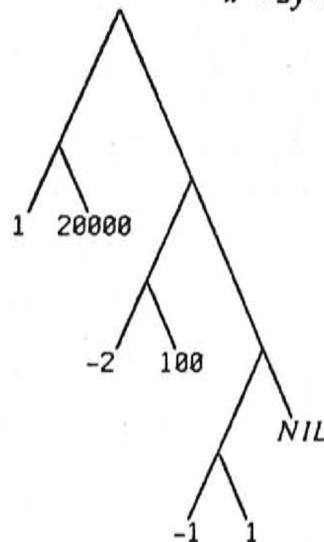
greaterp[x;y] = x>y.

In the construction of the 'sum' polynomial we will generate new terms by combining coefficients. So a constructor named *node* is needed. In terms of the latest representation *node* is defined as:

node[x;y] <= *cons*[x;y].

So here's a graphical representation of our favorite polynomial:

$x^2 - 2y - z$



Here's the algorithm:

```
polyadd[p;q] <=
   [null[p] →q; null[q] → p;
    eq[expo[first[p]];expo[first[q]]] →     [zerop[plus[coef[first[p]];coef[first[q]]]]
                                               → polyadd[rest[p];rest[q]];
                                             t → concat[node[plus[coef[first[p]];coef[first[q]]];
                                                                   expo[first[p]]];
                                                              polyadd[rest[p];rest[q]]]];
    greaterp[expo[first[p]];expo[first[q]]] → concat[first[p];polyadd[rest[p];q]];
    t → concat[first[q];polyadd[p;rest[q]]]]
```

Now for an explanation and example.

First we used some new LISP functions:

$$plus[x;y] = x + y$$
$$zerop[x] \Leftarrow eq[x;0]$$

polyadd is of the form: $[p_1 \rightarrow e_1; p_2 \rightarrow e_2; p_3 \rightarrow e_3; p_4 \rightarrow e_4; p_5 \rightarrow e_5]$

Case i: $p_1 \rightarrow e_1$ and $p_2 \rightarrow e_2$. If either polynomial is empty return the other.

Case ii: $p_3 \rightarrow e_3$. If the variable parts are equal then we can think about combining terms. However, we must check for cancellations and not include any such terms in our resultant polynomial.

Case iii: $p_4 \rightarrow e_4$ and $p_5 \rightarrow e_5$. These sections worry about the ordering of terms so that the resultant polynomial retains the ordering.

Here's an informal execution of *polyadd:*

$$
\begin{aligned}
&polyadd[x+y+z;x^2-2y-z] \\
&\quad = concat[x^2;polyadd[x+y+z;-2y-z]] \\
&\quad = concat[x^2;concat[x;polyadd[y+z;-2y-z]]] \\
&\quad = concat[x^2;concat[x;concat[node[1+-2;y];polyadd[z;-z]]]] \\
&\quad = concat[x^2;concat[x;concat[-y;polyadd[z;-z]]]] \\
&\quad = concat[x^2;concat[x;concat[-y;polyadd[(\ );(\ )]]]] \\
&\quad = concat[x^2;concat[x;concat[-y;(\ )]]] \\
&\quad = x^2+x-y
\end{aligned}
$$

Problems

1. Write an algorithm, ***polymult***, to perform the the multiplication of two polynomials.

2.5 Evaluation of polynomials

Though you are undoubtedly quite tired of looking at polynomials, there is at least one more operation which is usefully performed on such creatures. That operation is evaluation. Given an arbitrary polynomial, and values for any of the variables which it contains, we would like to

compute its value. First we will assume that the substitutions of values for variables has already been carried out. Thus we are dealing with polynomials of the form: $\Sigma^n_{i=1} p_i$ where p_i is a product of powers of constants. For example:

$$2^3 + 3*4^2 + 5.$$

This could be represented as:

(PLUS (EXPT 2 3) (PLUS (TIMES 3 (EXPT 4 2)) 5)).

We have taken this general representation because we have great expectations of generalizing the resulting algorithm.

We will now describe a LISP function, *value*, which will take such an S-expr representation and compute its value. Input to *value* will be numerals or lists beginning with either *PLUS, TIMES,* or *EXPT*. The value of a numeral is that numeral; to evaluate the other forms of input we should perform the operation represented. We must therefore assume that operations of addition, multiplication, and exponentiation exist. Assume they are named +, *, and ↑, respectively. What then should be the value of a representation of a sum? It should be the result of adding the value of the representations of the two summands or operands. That is, *value* is recursive. It should now be clear how to write *value*:

```
value[x] <=[isconstant[x] → x;
            issum[x] → +[value[arg1[x]];value[arg2[x]]];
            isprod[x] → *[value[arg1[x]];value[arg2[x]]];
            isexpt[x] → ↑[value[arg1[x]];value[arg2[x]]]]
```

where:

```
isconstant[x] <= numberp[x]
issum[x] <= eq[first[x];PLUS]
isprod[x] <= eq[first[x];TIMES]
isexpt[x] <= eq[first[x];EXPT]
```

Problems

1. Show how to extend *value* to handle binary and unary minus.

2. Write an algorithm *instantiate* which will take two arguments, one representing a set of variables and values, the other representing a polynomial. The algorithm is to return a representation of the polynomial which would result from substituting the values for the variables.

3. It would be nice if we could represent expressions like 2+3+4 as *(PLUS 2 3 4)* rather than *(PLUS (PLUS 2 3) 4)* or *(PLUS 2(PLUS 3 4))*; or represent 2*3*4+5+6 as *(PLUS (TIMES 2 3 4) 5 6)*.

Write a new version of *value* which can evaluate such n-ary representations of + and *.

More on polynomial evaluation

Though it should be clear that the current *value* function does perform the appropriate calculation, it should be equally clear that the class of expressions which *value* handles is not particularly powerful. We might wish to evaluate requests like:

A "What is the value of $x*y + 2*z$ when $x=4, y=2$, and $z=1$?"

Now the function *instantiate*, requested in problem 2 above, offers one solution: make a new copy of the representation of $x*y + 2*z$ with the variables physically replaced by their values. This would result in a representation of $4*2 + 2*1$, and this new expression is suitable fare for *value*. Computationally, this is a terrible solution. *instantiate* will go through the structure of the expression looking for instances of variables, and when located, will replace them with the appropriate values. *value* then goes through the structure of the resulting expression performing the evaluation. Clearly what is desireable is a function *value′* combining the two processes: the basic structure of *value′* is that of mild-mannered *value*, but when a variable, say *x*, is recognized inside *value′* then we would hope that it looks at a table like that expected by *instantiate*, finds *x* and returns the value associated with the entry for *x*.

Let's formalize our intuitions about *value′*. It will be a function of two arguments. The first will be a representation of a polynomial; the second will be a representation of the table of variables and values. You may have noticed that the original version of *value* in fact handles expressions which are not actually constant polynomials; $(2 + 3)*4$ for example. Since we will wish to apply our evaluation functions to more general classes of expressions we will continue, indeed encourage, this deception. Regardless of the class of expressions we wish to examine, it is the structure of the table which should be the first order of business. An appropriate table, *tbl*, will be a set of ordered pairs $<name_i, val_i>$; thus for the above example the table $\{<x, 4>, <y, 2>, <z, 1>\}$ would suffice. Following our dictum of abstraction and representation-independent programming, we will not worry about the representational problems of such tables. We will simply assume that "tables" are instances of an abstract data structure called <table>, and we will only concern ourselves for the moment with the kinds of operations we need to perform. We will need two selector functions: *name*, to select the variable-component of a table entry; and *val*, to select the value-component. A complete discussion of such a data structure would entail discussion of constructors and recognizers, and perhaps other functions, but for the current *value′* these two functions will suffice.

value′ will need a table-function, *locate*, to locate an appropriate variable-value entry. *locate* will be

a binary function, taking one argument, *x*, representing a variable; and one argument, *tbl*, representing a table. *locate* will match *x* against the *name*-part of each element in *tbl*; if a match is found then the corresponding *val*-part is returned. If no match is found then *locate* is undefined.

So far, little structure has been imposed on elements of <table>; tables are either empty or not; but if a table is non-empty then each element is a pair with recognizable components of name and value. However, the specification of algorithms to examine elements of <table> imposes more structure on our tables. If we were dealing with functions then a side condition to the effect that a table had no pairs with duplicate first elements would be sufficient. We, however, are dealing with algorithms and therefore must describe a method for locating elements, and soon must describe a method for adding elements.

Recursion is the only method we have for specifying *locate*, and recursion operates by decomposing a structure. Sets are notorious for their lack of structure; there is no order to the elements of a set. But if we are to write a LISP algorithm for *locate*, that algorithm will have to be recursive on the "structure" of *tbl*, and so we impose an ordering on the elements of that table. That is, we will represent tables as sequences. We know how to represent sequences in LISP: we use lists.

With this introduction, here's *locate* [42]:

```
locate[x;tbl] <=
        [eq[name[first[tbl]];x] → val[first[tbl]];
         t → locate[x;rest[tbl]]
         ]
```

The effect of *locate* is to find the first element of *tbl* which has a *name*-component which matches *x*. Having found that match, the corresponding *val*-part is returned. If there were other matches further along in the sequence *locate* would not see them. Other representations of tables are certainly possible. This representation will be useful in later applications.
And here's the new more powerful *value*′:

```
value′[x;tbl] <=
        [isconstant[x] → x;
         isvar[x] → locate[x;tbl];
         issum[x] → +[value′[arg₁[x];tbl];value′[arg₂[x];tbl]];
         isprod[x] → *[value′[arg₁[x];tbl];value′[arg₂[x];tbl]];
         isexpt[x] → ↑[value′[arg₁[x];tbl];value′[arg₂[x];tbl]]
         ]
```

Notice that *tbl* is carried through as an explicit argument to *value*′ even though it is only accessed when a variable is recognized. Notice too that much of the structure of *value*′ is quite repetitious;

[42] The obvious interpretation of *tbl* as a function implies that *locate* represents function application; i.e., *locate*[*x*;*tbl*] is *tbl*(*x*).

the lines which handle sums, products, and exponentiation are identical except for the function which finally gets applied to the evaluated arguments. That is, the basic structure of *value"* is potentially of broader application than just the simple class of polynomials. In keeping with our search for generality, let's pursue *value"* a little further.

What *value"* says is:

1. The value of a constant is that constant.

2. The value of a variable is the current value associated with it in the table.

3. The value of a function applied to arguments is the result of applying the function to the evaluated arguments. It just turns out that the only functions *value"* knows about are binary sums, products, and exponentiation.

Let's clean up *value"* a bit.

```
value"[x;tbl] <=
        [isconstant[x] → x;
         isvar[x] → locate[x;tbl];
         isfun_args[x] → apply[fun[x];eval_args[args[x];tbl]]
        ]
```

The changes are in the last branch of the conditional. We have a new recognizer, *isfun_args* to recognize function application. We have two new selector functions; *fun* selects the representation of the function -- sum, product, or power in the simple case; *args* selects the arguments or parameters to the function -- in this case all functions are binary. We have two new functions to define: *eval_args*, which is supposed to evaluate the arguments using *tbl* to find values for any of the variables; and *apply*, which is used to perform the desired operation on the evaluated arguments.

Again we are trying to remain as representation-free as possible: thus the generalization of the algorithm *value"*, and thus the care in picking representations for the data structures. We need to make another data structure decision now; when writing the function *eval_args*, we will be giving a recursive algorithm. This algorithm will be recursive on the structure of the first argument, which is a representation of the arguments to the function. In contrast to our position when writing the function *locate*, there is a natural structure on the arguments to a function: they form a sequence. That is *f[1;2;3]* is typically not the same as *f[3;2;1]* or *f* applied to any other permutation of {1, 2, 3}. Thus writing *eval_args* as a function, recursive on the sequence-structure of its first argument, is quite expected. Here is *eval_args*:

```
eval_args[args;tbl] <=
        [null[args] → ();
         t → concat[value"[first[args];tbl]; eval_args[rest[args];tbl]]
        ]
```

Notice that we have written *eval_args* without any bias toward binary functions; it will evaluate a sequence of arbitrary length, returning a sequence of the evaluated arguments.

There should be no real surprises in *apply*; it gets the representation of the function name and the sequence of evaluated arguments and does its job:

```
apply[fn; evargs] <=
        [issum[fn] → +[arg₁[evargs];arg₂[evargs]];
         isprod[fn] → *[arg₁[evargs];arg₂[evargs]];
         isexpt[fn] → ↑[arg₁[evargs];arg₂[evargs]]
        ]
```

Notice that if we should desire to recognize more functions then we need only modify *apply*. That would be a short-term solution, but if we wished to do reasonable computations we would like a more general function-definition facility. Such a feature would allow new functions to be defined during a computation; then when an application of that function were needed, the *value*-function would find that definition and apply it in a manner analogous to the way the pre-defined functions are applied. How far away are we from this more desirable super-*value*? Well *value*′ is already well-endowed with a mechanism for locating values; perhaps we can exploit this judiciously placed code. In what context would we be interested in locating function definitions? Here's an example:

B "What is the value of $f[4;2;1]$ when $f[x;y;z] <= x*y + 2*z$?"

Notice that if we have a means of recovering the definition of f, then we can reduce the problem to A of page 66. We will utilize the table-mechanism, and therefore will use *locate* to retrieve the definition of the function f. In our prior applications of *locate* we would find a constant as the associated value. Now, given the name f, we would expect to find the definition of the function. The question then, is how do we represent the definition of f? Certainly the body of the function, $x*y + 2*z$, is one of the necessary ingredients, but is that all? Given the expression $x*y + 2*z$ can we successfully compute $f[4;2;1]$? Not yet; we need to know the correspondence between the values $1, 2, 4$ and the variables, x, y, z. That information is present in our notation $f[x;y;z] <= ...$, and is a crucial part of the definition of f. That is, the order of the variables appearing after the function name is an integral part of the definition: $f[y;z;x] <= x*y +2*z$ defines a different function.

Since we are now talking about representations of functions, we are getting into the realm of abstract data structures. We have a reasonable understanding now of the essential components of such a representation. For our purposes, a function has three parts:

1. A name; f in the current example.

2. A variable list; $[x;y;z]$ here.

3. A body; $x*y + 2*z$ in the example.

We do not need a complete study of representations of functions. For our purposes we can assume a representation exists, and that we are supplied with three selectors to retrieve the components mentioned above.

1. *name* selects the name component from the representation. We have actually seen *name* before in the definition *locate* on page 67.

2. *varlist* selects the list of variables from the representation. We have already seen that the natural way to think about this component is as a sequence. Thus the name: *varlist*.

3. *body* selects the expression which is the content of the definition.

Given a function represented in the table according to these conventions, how do we use the information to effect the evaluation of something like *f[4;2;1]*? First *value'* will see the representation of *f[4;2;1]*; it should recognize an instance of function-application at the following line of *value'*:

isfun_args[x] → apply[fun[x];eval_args[args[x];tbl]]

This should cause an evaluation of the arguments and then pass on the hard work to *apply*.

Clever *apply* should soon realize that *f* is not the name of a known function. It should then extract the definition of *f* from the table; associate the evaluated arguments (*4, 2, 1*) with the variables of the variable list (*x, y, z*), making a new table with name-value pairs (<*x*, *4*>, <*y*, *2*>, <*z*, *1*>). Now we are back to the setting of problem A of page 66. We should ask *value'* to evaluate the *body*-component of the function using the new *tbl*. This works fine for *x*, *y*, and *z*; within the evaluation of the body of *f* we will find the right bindings for these variables. But we might also need some information from the original *tbl*. The evalaution of the body of *f* might entail the application of some function definition present in *tbl*. For example, the representation of

"what is *g[2]* where *g[x] <= x+s[x]*; and *s[x] <= x*x*?"

Within the body of *g* we need the definition of *s*. Therefore, instead of building a new table we will add the new bindings to the front of the old table. Since *locate* begins its search from the front of the table we will be assured of finding the new bindings; since the old table is still accessible we are assured of finding any necessary previous bindings.

Now we should be able to go off and create a new *value''*. Looking at the finer detail of *value'* and *apply*, we can see a few other modifications need to be made. *apply'* will use *value''* to *locate* the function definition and thus *tbl* should be included as a third argument to *apply'*. That is, inside *apply'* we will have:

isfun[fn] → apply'[value''[fn;tbl];eargs;tbl];

After *value''* has done its work, this line (above) will invoke *apply'* with a function definition as first argument. We'd better prepare *apply'* for such an eventuality with the following addition:

```
isdef[fn] → value''[body[fn];newtbl[varlist[fn];eargs;tbl]];
```

What does this incredible line say? It says

> "Evaluate the body of the function using a new table manufactured from the old table by adding the pairings of the elements of the variable list with the evaluated arguments."

It also says we'd better write *newtbl*. This function will be making a new table by adding new name-value pairs to an existing table. So we'd better name a constructor to generate a new name-value pair:

mkent is the constructor to make new entries. It will take two arguments: the first will be the name, the second will be the value.

Since we have assumed that the structure of tables, variable-lists, and calling sequences to functions are all sequences, we will write *newtbl* assuming this representation.

```
newtbl[vars;vals;tbl] <=
        [null[vars] → tbl;
         t → concat[mkent[first[vars];first[vals]];newtbl[rest[vars];rest[vals];tbl]]
        ]
```

And finally here's the new *value''-apply'* pair:

```
value''[x;tbl] <=
        [isconstant[x] → x;
         isvar[x] → locate[x;tbl];
         isfuname[x] → locate[x;tbl]; [43]
         isfun_args[x] → apply'[fun[x];eval_args[args[x];tbl];tbl]
        ]

apply'[fn;evargs;tbl] <=
        [issum[fn] → +[arg1[evargs];arg2[evargs]];
         isprod[fn] → *[arg1[evargs];arg2[evargs]];
         isexpt[fn] → ↑[arg1[evargs];arg2[evargs]];
         isfun[fn] → apply'[value''[fn;tbl];evargs;tbl];
         isdef[fn] → value''[body[fn];newtbl[varlist[fn];evargs;tbl]];
        ]

eval_args[args;tbl] <=
        [null[args] → ()
         t → concat[value'[first[args];tbl]; eval_args[rest[args];tbl]]
        ]
```

---

[43] We have two separate cases here to emphasize the difference between a simple variable and a variable used as a function name.

Let's go through a complete massaging of B of page 69. As before we will use $\Re$ as a mapping from expressions to representations. Thus we want to pursue:

$$value''[\Re[\![\ f[4;2;1]\ ]\!];\ \Re[\![\{\ \langle f, [[x;y;z]\ x*y + 2*z] \rangle\ \}]\!]].$$

Before we begin, let us denote the initial symbol table, $\Re[\![\{\ \langle f, [[x;y;z]\ x*y + 2*z]\rangle\ \}]\!]$ as *init*. This will simplify many of the equations. Notice that our representation of f in *init* has associated the variable list $[x;y;z]$ with the body of the function. Thus *locate*, operating on this table with the name f, will return a representation of $[[x;y;z]\ x*y + 2*z]$.

isfun_args should be satisfied and thus the computation should reduce to:

$$apply'[fun[\ \Re[\![\ f[4;2;1]\ ]\!]\ ];eval_args[args[\ \Re[\![\ f[4;2;1]\ ]\!]\ ];\ init\ ] \text{ or:}$$

$$apply'[\ \Re[\![\ f\ ]\!]\ ;eval_args[\ \Re[\![\ [4;2;1]\ ]\!];\ init];\ init\ ]$$

eval_args will build a sequence of the evaluated arguments: $(4, 2, 1)$, resulting in:

$$apply'[\ \Re[\![\ f\ ]\!]\ ;(4, 2, 1)\ ;\ init\ ]$$

apply' should decide that f satisfies *isfun* giving:

$$apply'[\ value''[\ \Re[\![\ f\ ]\!]\ ;\ init\ ];\ (4, 2, 1)\ ;\ init\ ]$$

The computation of: value''$[\ \Re[\![\ f\ ]\!]\ ;\ init\ ]$ is interesting: *value* should believe that f is a function name and therefore *locate* will be called and retrieve the definition.

$$apply'[\ \Re[\![\ [[x;y;z]\ x*y + 2*z]\ ]\!]\ ;\ (4, 2, 1)\ ;\ init\ ] \text{ should be the result.}$$

This first argument should not make *apply'* unhappy. It should realize that *isdef* is satisfied, and thus:

$$value''[body[\ \Re[\![\ [[x;y;z]\ x*y + 2*z]\ ]\!]\ ];newtbl[varlist[\ \Re[\![\ [[x;y;z]\ x*y + 2*z]\ ]\!]\ ];(4,2,1);init]] \text{ or:}$$

$$value''[\ \Re[\![\ [x*y + 2*z]\ ]\!]\ ;newtbl[\ \Re[\![\ [x;y;z]\ ]\!]\ ;(4,2,1);init]]$$

after *body* and *varlist* are finished. But $\Re[\![\ [x;y;z]\ ]\!]$ is $(\Re[\![\ x\ ]\!], \Re[\![\ y\ ]\!], \Re[\![\ z\ ]\!])$, and therefore the computation of *newtbl* will build a new table with entries for x, y, and z on the front:

$$\Re[\![\{\ \langle x, 4\rangle, \langle y, 2\rangle, \langle z, 1\rangle, \langle f, [[x;y;z]\ x*y + 2*z]\rangle\ \}]\!].$$

Thus we call *value''* with:

$$value''[\ \Re[\![\ [x*y + 2*z]\ ]\!];\ \Re[\![\{\ \langle x, 4\rangle, \langle y, 2\rangle, \langle z, 1\rangle, \langle f, [[x;y;z]\ x*y + 2*z]\rangle\ \}]\!]\ ].$$

Now we're back at problem A of page 66.

Time to take stock

We have written a reasonably sophisticated algorithm here. It covers evaluation of a large class of arithmetic functions. We should examine the results quite carefully.

First notice that we have written the algorithm with almost no concern for representation. We assume that representations are available for such varied things as arithmetic expressions, tables, calls on functions, and even function definitions. Very seldom did we commit ourselves to anything close to a concrete representation, and only then with great reluctance. It was with some sadness that we imposed a sequencing on elements of tables. Variable lists and calling sequences were not as traumatic; we claimed their natural structure was a sequence. As always, if we wish to run these programs on a machine we must supply some representations, but even then the representations will only interface with our algorithms at the constructors, selectors and recognizers.

We have made some more serious representational decisions in the structure of the algorithm. We have encoded a version of the CBV-scheme of page 14. We have seen what kinds of difficulties that can get us into. We will spend a large amount of time in Section 3 discussing the problems of evaluation [44].

Finally, our decisions on the data structures and the algorithms were not made independently. For example, there is strong interaction between our representation of tables and the algorithms, *locate* and *newtbl* which manipulate them. We should ask how much of this interaction is inherent and how much is gratutious. For example we have remarked that our representation can have pairs with duplicate first elements. It is the responsibility of *locate* to see that we find the expected pair. If we wrote *locate* to search from right to left, we could get the wrong pair. We could write *newtbl* to be more selective; it could manufacture a table without such duplications:

```
newtbl[vars;vals;tbl] <=
        [null[tbl] →     [null[vars] → ();
                          t → concat [mkent[first[vars];first[vals]];
                                      newtbl[rest[vars];rest[vals];( )]]
         member[name[first[tbl]];vars] → newtbl[vars;vals;rest[tbl]]
         t → concat[first[tbl];newtbl[vars;vals;rest[tbl]]]
        ]
```

[44] A second decision was implied in our handling of function definitions; namely we bound the function name to a structure consisting of the variable list and the function body. This representation gives the expected result in most cases, but involves one of the more problematic areas of programming languages: how do you find the bindings of variables which do not appear in the current variable list? Function names belong in this category. Such variables are called free variables. The scheme proposed in this section finds the binding which is current when the function was applied. The programming language, ALGOL, advocates finding the binding which was current when the function was defined.

This version of *newtbl* requires much more computation than the alternative. Its advantage is that the "set"-ness of symbol tables is maintained. Luckily the "set" property is one which we need not depend on for our algorithms. Indeed we will frequently depend on the obverse property: that the previous values of variables can be found further along in the sequence.

The main point of this example however is to impress on you the importance of writing at a sufficiently high level of abstraction. We have produced a non-trivial algorithm which is clear and concise. If it were desirable to have this algorithm running on a machine we could code it and its associated data structure representations in a very short time. In a very short time we will be able to run this algorithm on a LISP machine.

2.6 The great mothers

The following problems are written (intentionally) with a great deal of the representation built into them. They are truly ugly but should be done anyway.

I. The Great Mother of All Functions!!! (*tgmoaf*)

```
tgmoaf[x] <=    [isindiv[x] → [eq[x ;T] → t;
                               eq[x;NIL] → f;
                               t → TRYAGAINNEXTWEEK];
                 eq[first[x];QUOTE] → second[x];
                 eq[first[x];CAR] → car[tgmoaf[second[x]]];
                 eq[first[x];CDR] → cdr[tgmoaf[second[x]]];
                 eq[first[x];CONS] → cons[tgmoaf[second[x]];tgmoaf[third[x]]];
                 eq[first[x];ATOM] → atom[tgmoaf[second[x]]];
                 eq[first[x];EQ] → eq[tgmoaf[second[x]];tgmoaf[third[x]]];
                 t → TRYAGAINNEXTWEEK]
```

Evaluate the following:

1. *tgmoaf[T]*
2. *tgmoaf[A]*
3. *tgmoaf[(CAR (QUOTE (A . B)))]*
4. *tgmoaf[(CDR (QUOTE (A B)))]*
5. *tgmoaf[(EQ (CAR (QUOTE (A . B))) (QUOTE A))]*
6. *tgmoaf[(EQ (CAR (QUOTE (A . B))) A)]*
7. *tgmoaf[(ATOM (CAR (QUOTE (A B))))]*

II. The Great Mother of All Functions Revisited!!!(*tgmoafr*)

```
tgmoafr[x] <=   [isindiv[x] → [eq[x;T] → t;
                               eq[x;NIL] → f;
                               t → TRYAGAINNEXTWEEK];
                 eq[first[x];QUOTE] → second[x];
                 eq[first[x];CAR] → car[tgmoafr[second[x]]];
                 eq[first[x];CDR] → cdr[tgmoafr[second[x]]];
                 eq[first[x];CONS] → cons[tgmoafr[second[x]];tgmoafr[third[x]]];
                 eq[first[x];ATOM] → atom[tgmoafr[second[x]]];
                 eq[first[x];EQ] → eq[tgmoafr[second[x]];tgmoafr[third[x]]];
                 eq[first[x];COND] → evcond[rest[x]];
                 t → TRYAGAINNEXTWEEK]
```

evcond[*x*] <= [*tgmoafr*[*first*[*first*[*x*]]] → *tgmoafr*[*second*[*first*[*x*]]];
t → *evcond*[*rest*[*x*]]]

Evaluate the following:

1. *tgmoafr*[*T*]
2. *tgmoafr*[(*CDR* (*QUOTE* (*A B*)))]
3. *tgmoafr*[(*EQ* (*CAR* (*QUOTE* (*A* . *B*))) (*QUOTE A*))]
4. *tgmoafr*[(*COND* ((*EQ* (*CAR* (*QUOTE* (*A* . *B*))) (*QUOTE A*))(*QUOTE FOO*)))]
5. *tgmoafr*[(*COND* ((*ATOM* (*QUOTE* (*A*)))(*QUOTE FOO*)) (*T*(*QUOTE BAZ*)))]

Coming soon: **Son of Great Mother !!**

2.7 Another Respite

We have again reached a point where a certain amount of reflection would be good for the soul. Though this is not a programming manual we would be remiss if we did not attempt to analyze the style which we have tried to exercise when writing programs.

1. Write the algorithm in an abstract setting; do not muddle the abstract algorithm with the chosen representation. If you follow this dictum your LISP programs will never use *car, cdr, cons,* and *atom*. All instances of these LISP primitives will be banished to small subfunctions which manipulate representations.

2. When writing the abstract program, do not be afraid to cast off difficult parts of the implementation to subfunctions. Remember that if you have trouble keeping the details in mind when writing the program, then the confusion involved in reading the program at some later time will be overwhelming. Once you have convinced yourself of the correctness of the current composition, then worry about the construction of the subfunctions. Seldom does the process of composing a program flow so gently from top-level to specific representation. Only the toy programs are easy; the construction of the practical program will be confusing, and will require much rethinking. But bring as much structure as you can to the process.

3. From the other side of the question, don't be afraid to look at specific implementations, or specific data-structure representations before you begin to write. There is something quite comforting about a "real" data structure. Essentially data structures are static objects [45], while programs are dynamic objects. A close look at a possible representation may get you a starting point and as you write the program it will become clear when you are depending on the specific representation and when you are just using properties of an abstract data structure.

[45] At least within the program presently being constructed.

Perhaps the more practical reader is ovecome by the inefficiencies inherent in these proposals. Two answers: first, "inefficiency" is a very etherial concept. Like "structured programming", it is difficult to define but recognizable when it occurs. Hardware and software development has been such that last year's "inefficiency" is this year's *passe* programming trick. But even at a more topical level, much of what seems inefficent can now be straightened out by a compiler (see Section). Frequently compilers can do very clever optimizations to generate efficient code. It is better to leave the cleverness to the compiler, and the clarity to the programmer.

Second, the problems in programming are not those of efficiency. They are problems of correctness. How do you write a program which works? Until practical tools are developed for proving correctness it is up to the programmer to certify his programs. Any methodology which can aid the programmer will be most welcome. Clearly, the closer you can write the program to your intuition, the less chance there is for error. This was one of the reasons for developing high-level languages. But do not forget that the original motivation for such languages was a convenient notation for expressing numerical problems, that is, for writing programs to express ideas which have already had their juices extracted as mathematical formulas. With data structures, we are able to formalize a broader range of domains, expressing our ideas as data structure manipulations rather than as numerical relationships.

What kinds of errors are prevelant in data structure programming? At least two kinds: errors of omission -- misunderstanding of the basic algorithm; and errors of commission -- errors due to misapplied cleverness in attempting to be efficient.

The occurrences of errors of omission can be lessened by presenting the user with programming constructs which are close to the intuited algorithm. Such constructs include control structures, data structures, and representations for operations.

Errors of commission comprise the great majority of the present day headaches. It is here that programming style can be beneficial: keep the representation of the data structures away from the description of the algorithm; write concise abstract programs, passing off responsibilities to subfunctions. Whenever a definition of "structured programming" is arrived at, this advice on programming style should be included.

Before closing this discussion on the philosphy of LISP programming, we can't but note that the preceding section, **The Great Mothers**, completely ignored our good advice. This would be a good time for the interested reader to abstract the *tgmoaf* algorithm from the particular data representation. This detective work will be most rewarding.

Problems

I. Write an abstract version of *tgmoaf*.

2.8 Proving properties of programs

People are becoming increasingly aware of the importance of giving convincing arguments for such things as the correctness or equivalence of programs. These are both very difficult enterprises. We will only sketch a proof of a simple property of two programs and leave others as problems for the interested reader. How do you go about proving properties of programs? In Section 1.9 we noted certain benefits of defining sets using inductive definitions. First, there was a natural way of thinking about the construction of an algorithm over that set. We have exploited that observation in our study of LISP programming. What we need recall here is the observation that inductive style proofs (see PRF on page 40) are valid forms of reasoning over such domains. Since we in fact defined our data structure domains in an inductive manner, it seems natural to look for inductive arguments when proving properties of programs. This is indeed what we do; we perform induction on the structure of the elements in the data domain.

For example, using the definition of *append* given on page 44 and the definition of *reverse* given on page 45, we wish to show that:

append[*reverse*[y];*reverse*[x]] = *reverse*[*append*[x;y]],

for any lists, x, and y. The induction will be on the structure of x.

Basis: x is ( ).

We must thus show: *append*[*reverse*[y];( )] = *reverse*[*append*[( );y]]

But: *reverse*[*append*[( );y]] = *reverse*[y] by the def. of *append*.

We now establish the stronger result: *append*[z;( )] = z [46]

Basis: z is ( ).

Show *append*[( );( )] = ( ). Easy.

Induction step: Assume the lemma for lists, z, of length n;

Prove: *append*[*concat*[x;z];( )] = *concat*[x;z].

Since *concat*[...] is not ( ), then applying the definition of *append* says we must prove: *concat*[x;*append*[z;( )]] = *concat*[x;z].

But our induction hypothesis is applicable since z is shorter than *concat*[x;z].

Our result follows.

So the Basis for our main result is established.

[46] In the following proof several intermediate steps have been ommitted.

Induction step: Assume the result for lists, z, of length n;
Prove:

(1) *append[reverse[y];reverse[concat[x;z]]] = reverse[append[concat[x;z],y]]*

Using the definition of *reverse* on the LHS of (1) gives:

(2) *append[reverse[y];append[reverse[z];list[x]]].*

Using the definition of *append* on the RHS of (1) gives:

(3) *reverse[concat[x;append[z;y]]].*

Using the definition of *reverse* on (3) gives:

(4) *append[reverse[append[z;y]];list[x]].*

Using our induction hypothesis on (4) gives:

(5) *append[append[reverse[y];reverse[z]];list[x]]*

Thus we must establish that (2) = (5).
But this is just an instance of the associativity of *append*:

append[x;append[y;z]] = append[append[x;y];z]. (See problem I, below.)

The structure of the proof is analogous to proofs by mathematical induction in elementary number theory. The ability to perform such proofs is a direct consequence of our careful definition of data structures. Examination of the proof will show that there is a close relationship between what we are inducting on in the proof and what we are recurring on during the evaluation of the expressions. A program written by Boyer and Moore has been reasonably successful in generating proofs like the above by exploiting this relationship.

Problems

I. Prove the associativity of *append.*

II Analysis of the above proof shows frequent use of other results for LISP functions. Fill in the details. Investigate the possibility of formalizing this proof, showing what axioms are needed.

III Show the equivalence of *fact* (page 41) and $fact_1$ (page 43).

IV Show the equivalence of *length* and $length_1$ (page 43).

V Using the definition of *reverse*, given on page 44, prove:

reverse[reverse[x]] = x.

SECTION 3

EVALUATION OF LISP EXPRESSIONS

"... I always worked with programming languages because it seemed to me that until you could understand those, you really couldn't understand computers. Understanding them doesn't really mean only being able to use them. A lot of people can use them without understanding them. ..."

Christopher Strachey, *The Word Games of the Night Bird.*

3.1 Introduction

In the previous sections of this text we have talked about some of the schemes for evaluation. We have done so rather informally for LISP; we have been more precise about evaluation of simple arithmetic expressions. Section 2.5 discusses this in some detail. We shall now look more closely at the informal process which we have been using in the evaluation of LISP expressions. This is motivated by at least two desires.

First, we want to run our LISP programs on a machine. To do so requires the implementation of some kind of translator to turn LISP programs into instructions which can be carried out by a conventional machine. We will be interested in the structure of such implementations. Any implementation of LISP must be grounded on a precise, and clear understanding of what LISP-evaluation entails. Indeed, a deep understanding of evaluation is a prerequisite for implementation of any language. The question of evaluation cannot be sidestepped by basing a language on a compiler. A compiler must produce code which when executed, simulates the evaluation process. There is no escape.

Our second reason for pursuing evaluation involves the question of programming language specification. This title covers a multitude of sins. At a practical level we want a clean, machine independent, "self-evident" description of a language specification, so that the agony involved in implementing the design can be minimized. At a more abstract level, we should try to understand just what is specified when we design a language. Are we specifying a single machine, a class of machines, a class of mathematical functions. Just what is a language? The syntactic specification of languages is reasonably well established, but syntax is only the tip of the iceberg. Our study of LISP will address itself to the deeper problems of semantics, or meaning, of languages.

Before we address the direct question of LISP evaluation, we should perhaps wonder aloud about the efficacy of studying languages in the detail which we are proposing. As computer scientists we

should be curious about the structure of programming languages because we must understand our tools -- our programming languages. People who simply wish to use computers as tools need not care about the structure of languages. Indeed they usually couldn't care less about the inner workings of the language; they only want languages in which they can state their problems in a reasonably natural manner. They want their programs to run and get results. They are interested in the output and seldom are interested in the detailed process of computation. For a simple analogy, consider the field of mathematics. The practicing mathematician uses his tools -- proofs -- in a similar manner to the person interested in computer applications. He seldom needs to examine questions like "what is a proof?" He does not analyze his tools. However, it must be pointed out that not so many years ago such questions indeed were raised, and for good reason. Some common forms of reasoning were shown to lead to contradictions unless care was taken.

Our position is more like that of the foundations of mathematics; there the tools of mathematics are studied and held up to the light. Mathematics has flourished because of it. Though our expectations are not quite that presumptuous, we do expect that programming language design cannot help but be improved.

Our study of language implementation will proceed from the abstract to the concrete. Each level will intimately involve the study of data structures. The current chapter will be the most abstract, building a precise high-level description of an evaluation scheme for LISP. In fact, the discussion is much more general than that of LISP; the text addresses itself to problem areas in the design of any reasonably sophisticated language. In subsequent chapters we probe beneath the surface of this high-level description and discuss common ways of implementing the necessary data structures.

But how can we begin to understand LISP evaluation? In Section 2.5 we made a beginning, giving an algorithm for a subset of the computations expressible in LISP. This subset covered evaluation of some simple arithmetic expressions. From our earliest grade school days we have had to evaluate simple arithmetic expressions. Later, in algebra we managed to cope with expressions involving function application. Most of us survived the experience. We should now try to understand the processes we used in these simple arithmetic cases, doing our examination at the most mechanical level. The basic intent of the algorithm is fixed: evaluate the expression; but within that general constraint we often have several distinct alternative methods. Those places at which we have choice should be remembered. We will make reasonable choices so that the process becomes deterministic and proceed. Later, we should reflect on what effect our choices had on the resulting scheme. For example, recall the discussion of the representation of symbol tables on page 73. We had several options, but picked one which seemed to satisfy our intuitions and was reasonably efficient. But we should subject that decision to close scrutiny: does it really fulfill our expectations and is it efficient? In absence of absolute standards, these questions are usually answered by examining the behavior of the algorithm.

The first thing to note in examining simple arithmetic examples is that **nothing** is really said about the process of evaluation. When asked to evaluate $(2*3)+(5*6)$ we never specified which summand was to be evaluated first. Indeed it didn't matter here. $6+(5*6)$ or $(2*3)+30$ both end up 36. Does it ever matter? "+" and "*" are examples of arithmetic functions; can we always leave the order of evaluation unspecified for arithmetic functions? How about evaluation of arbitrary

functional expressions? If the order doesn't matter, then the specification of the evaluation process becomes much simpler. If it does matter then we must know why and where.

We have seen that the order of evaluation can make a difference in LISP. On page 19 we saw that order of evaluation in conditional expressions can make a difference. On page 14 we saw that CBV, LISP's computational interpretation of function evaluation, requires some care. Since we are using CBV we must make some decision regarding the order of evaluation of the arguments to a function call, say $f[t_1;t_2; ...t_n]$. We will assume that we will evaluate the arguments from left to right.

Consider the example due to J. Morris:

$$f[x;y] <= [x = 0 \rightarrow 0;\ t \rightarrow f[x-1;f[y-2;x]]].$$

Evaluation of $f[2;1]$ will terminate if we always evaluate the outermost occurrence of f. Thus:

$$f[2;1] = f[1;f[-1;2]] = f[0;f[f[-1;2]-2;1]] = 0;$$

However if we evaluate the innermost occurrences [47] first, the computation will not terminate:

$$f[2;1] = f[1;f[-1;2]] = f[1;f[-2;f[0;-1]]] = f[1;f[-2;0]] = ...$$

So the choice of evaluation schemes is, indeed, fraught with peril. The evaluation scheme, CBV, which we chose is called **call-by-value** and is an inside-out style of evaluation, meaning that we operate on the subexpressions before tackling the main expression. Alternative proposals exist and have their merits; call-by-name evaluation is another well-known scheme. We advertised this outside-in scheme on page 14 as CBN. From a computational perspective, we can live with call-by-value, though we know the use of such a rule may lead to non-terminating computations when call-by-name would run to completion.

Intuitively, call-by-value says: evaluate the arguments to a function before you attempt to apply the function definition to the arguments. Let's look at a simple arithmetic example. Let $f[x;y]$ be $x^2 + y$ and consider $f[3+4;2*2]$. Then call-by-value says evaluate the arguments, getting 7 and 4; associate those values with the formal parameters of f (i.e. 7 with x and 4 with y) and then evaluate the body of f resulting in $7^2 + 4 = 53$. This is the scheme we captured in Section 2.5.

Call-by-name says pass the **unevaluated** actual parameters to the function, giving $(3+4)^2 + 2*2$ which also evaluates to 53. Call-by-name is essentially a "substitution followed by simplification" system of computation. We will say more about call-by-name and other styles of evaluation in Section . Most of this section will be restricted to call-by-value.

---

[47] The notions of "innermost" and "outermost" evaluation need to be slightly embellished for general function application. If the chosen application has several arguments, then we must specify an order for their evaluation. Thus terms like "outermost-leftmost" and "innermost-rightmost" occur. For example, the LISP scheme is an instance of "innermost-leftmost" evaluation.

If you look at the structure of *value''* and *apply'* beginning on page 71 you will see that they encode the CBV philosophy, are recursive, and have an intended interpretation which goes something like this:

1. If the expression is a constant then the value of the expression is that constant. (The value of *3* is *3* [48]).

2. If the expression is a variable then see what the current value associated with that variable is. Within the evaluation of, say, $f[3;4]$ where $f[x;y] \Leftarrow x^2 + y$ the current value of the variable x is *3*.

3. The only other kind of arithmetic expression that we can have is a function name followed by arguments, for example $f[3;4]$. In this case we first evaluate the arguments [49] and then apply the definition of the function to those evaluated arguments. How do we apply the function definition to the evaluated arguments? We associate or bind the formal parameters (or variables) of the definition to the values of the actual parameters. We then evaluate the body of the function in this new environment. Notice that we do not explicitly substitute the values for the variables which appear in an expression. We simulate substitutions by table lookup.

A moments reflection on the informal evaluation process we use in LISP should convince us of the plausibility of describing LISP evaluation at a similar level of precision. First, if the LISP expression is a constant, then the value of the expression is that constant. What are the constants of LISP? They're just the S-exprs. Thus the value of *(A . B)* is *(A . B)*, just like the value of *3* is *3*. Variables and functional applications appear in LISP and are handled as described above in 2. and 3.. The additional artifact of LISP which we have to include in a discussion of evaluation is the conditional expression. But clearly its evaluation can also be precisely specified. We did so on page 18. So, in more specific detail, here is some of the structure of the LISP evaluation mechanism:

1. If the expression to be evaluated is a constant then the value is that constant.

2. If the expression is a variable find its value in the current environment.

3. If the expression is a conditional expression then it is of the form $[p_1 \rightarrow e_1; p_2 \rightarrow e_2; \ldots p_n \rightarrow e_n]$. Evaluate it using the semantics defined on page 18.

4. If the expression is of the form: $f[t_1;t_2; \ldots ;t_n]$ then:

---

[48] We are ignoring the distinction between the numeral *3* and the **number** *3*.

[49] Here we are using the evaluation process recursively.

a. Evaluate the arguments $t_1, t_2, \ldots ,t_n$ from left to right.

b. Find the definition of the function, f.

c. Associate the evaluated arguments with the formal parameters in the function definition.

d. Evaluate the body of the function, while remembering the values of the variables.

We have seen (Section 2.5) that we can transcribe a simple kind of arithmetic evaluation into a recursive LISP program. That program operates on a representation of the expression and produces the value. Most of our work in that example was done without giving explicit details of the representation. However when we discussed the representation of simple differentiation in Section 2.3 we showed a detailed representation.

We have demonstrated an informal, but reasonably precise, evaluation scheme for LISP; our discussion is ready for a more formal development. It should be clear that we could write a LISP function representing the evaluation process provided that we can find a representation for LISP expressions as S-expressions. This mapping, $\mathfrak{R}$, of LISP expressions to S-exprs is our first order of business. We will accomplish this mapping by using an extension of the scheme introduced in Section 2.3. We plan to expend some effort in describing a specific representation for LISP expressions for two reasons. First, though abstraction is a most desirable attribute, we must reconcile our abstraction with reality; our programs must run. The second point is that the representation we pick will have a very close relationship to the way we present programs to the machine. So we will be careful and thorough in describing the mapping and you should be conscientious in your understanding of that mapping. Once that representation is given we will produce a LISP algorithm which describes the evaluation process used in LISP.

This process of mapping LISP expressions onto S-exprs and writing a LISP function to act as an evaluator may seem a bit incestuous; indeed, the rationale for doing any of this may not be apparent. Patience, please. The mapping is no more obscure than that in the polynomial evaluation or differentiation examples. It is just another instance of the diagram of page 50, only now we are applying the process to LISP itself. The effect is to force us to make precise exactly what is meant by LISP evaluation. This precision will have many important ramifications.

Also, we've been doing evaluation of S-expr representations of LISP expressions already. The **great mother of all functions** is exactly the evaluation mechanism for the LISP primitive functions and predicates, *car, cdr, cons, atom* and *eq* when restricted to functional composition and constant arguments. The **great mother revisited** is the extension to conditional expressions.

In the next section we will give a typical mapping of LISP expressions to elements of <sexpr>. But remember that we should attempt to keep the knowledge of the representation out of the structure of the algorithm. Let's stop for some examples of translating LISP functions into S-expr notation.

3.2 S-expr translation of LISP expressions

We will go through the list of LISP constructs, describing the effect of the representational map, $\mathfrak{R}$, and give a few examples applying $\mathfrak{R}$.

We will represent numerals just as numerals, e.g.:

$$\mathfrak{R}[\![\langle\text{numeral}\rangle]\!] = \langle\text{numeral}\rangle$$

$$\mathfrak{R}[\![2]\!] = 2$$

We will translate identifiers to their upper-case counterpart. Thus:

$$\mathfrak{R}[\![\langle\text{identifier}\rangle]\!] = \langle\text{literal atom}\rangle$$

$$\mathfrak{R}[\![x]\!] = X$$

$$\mathfrak{R}[\![y2]\!] = Y2$$

$$\mathfrak{R}[\![car]\!] = CAR$$

Now we've got a problem: We wish to map arbitrary LISP expressions to S-expressions. The LISP expression x translates to X. X is itself a LISP expression (a constant); it must also have a translation. We must be a little careful here. When we write Son of Great Mother we will give it an S-expr representation of a form to be evaluated. We might give it the representation of $car[x]$ in which case the value computed will depend on the current value bound to x. We might also give the representation of $car[X]$; in this case we should expect to be presented with $\perp$ or an error message. Or, for example, some function *foo* we wish to write may return the S-expr representation of a LISP form as its value. Say *foo*[*1*] returns the representation of $car[x]$ and *foo*[*2*] returns the representation of $car[X]$. We must be able to distinguish between these representations. That is, given the representation, there should be exactly one way of interpreting it as a LISP expression. The mapping must be 1-1. So we must represent x and X as different S-exprs. The translation scheme we pick is: for any S-expression α, its translation is $(QUOTE\ \alpha)$.

$$\mathfrak{R}[\![\langle\text{sexpr}\rangle]\!] = (QUOTE\ \langle\text{sexpr}\rangle)$$

For example:

$$\mathfrak{R}[\![X]\!] = (QUOTE\ X)$$

$$\mathfrak{R}[\![(A\ .B)]\!] = (QUOTE\ (A\ .\ B))$$

$$\mathfrak{R}[\![QUOTE]\!] = (QUOTE\ QUOTE)$$

We must also show how to map expressions of the form $f[e_1\ ; ...\ ;e_n]$ onto S-exprs. We have already seen one satisfactory mapping for functions in prefix form in Section 2.3. We will use that mapping, called Cambridge Polish [50], here. That is:

$$\mathfrak{R}[\![f[e_1;e_2;\ ...;e_n]]\!] = (\ \mathfrak{R}[\![f]\!]\ \mathfrak{R}[\![e_1]\!]\ \mathfrak{R}[\![e_2]\!]\ ...\ \mathfrak{R}[\![e_n]\!]\ )$$

---

[50] The name, Cambridge Polish, is derived from two sources: Cambridge, since M.I.T. is in Cambridge Massachusetts, and McCarthy was at M.I.T. while developing his ideas; Polish, since the representation is a dialect of a notation developed by a school of Polish logicians.

Examples:

$$\Re[\![car[x]]\!] = (\Re[\![car]\!] \Re[\![x]\!]) = (CAR\ X)$$
$$\Re[\![car[X]]\!] = (\Re[\![car]\!] \Re[\![X]\!]) = (CAR\ (QUOTE\ X))$$
$$\Re[\![cons[cdr[(A\ .B)];x]]\!] = (CONS\ (CDR\ (QUOTE\ (A\ .\ B)))\ X)$$

The $\Re$-mapping must handle conditional expressions. A conditional is represented as a list whose first element is *COND* and whose next *n* elements are representations of the p_i-e_i pairs. The $\Re$-map of such pairs is a list of the $\Re$-maps of the two elements:

$$\Re[\![[p_1 \to e_1; \ldots ;p_n \to e_n]]\!] = (COND\ (\Re[\![p_1]\!] \Re[\![e_1]\!]) \ldots (\Re[\![p_n]\!] \Re[\![e_n]\!]))$$

An example:

$$\Re[\![[atom[x] \to 1;\ q[y] \to X]]\!] = (COND\ ((ATOM\ X)\ 1)\ ((Q\ Y)\ (QUOTE\ X)))$$

Notice that *(COND ...)* and *(QUOTE ...)* look like translations of function applications of the form *cond*[...] and *quote*[...]. However since we expect application to be performed using call-by-value, we must handle these constructs in a special manner. Indeed, note that *quote*[α] stands for $\Re[\![\alpha]\!]$, and that the "arguments" to *cond* are not to be interpreted as some kind of function applications. For example, *COND ((ATOM X) 1) ...)* doesn't represent *cond*[*atom*[x][*1*]; ...].

Finally, the translations of the truth values t and f will be *T* and *NIL*, respectively.

$$\Re[\![\mathfrak{t}]\!] = T$$
$$\Re[\![\mathfrak{f}]\!] = NIL$$

You might have noticed that these last two applications of the $\Re$-mapping have the potential to cause trouble. They will spoil the 1-1 property of $\Re$:

$$\Re[\![t]\!] = T$$
$$\Re[\![nil]\!] = NIL$$

The usual way to escape from this difficulty is to outlaw *t* and *nil* as LISP variables [51].

Perhaps our concern for the $\Re$-mapping's properties appears heavy-handed where a simple solution seems apparent: t is t and *t* is *t*; when we want the truth value we write t and when we want the variable we write *t*. The problem is that when we write programs in a format which a machine version of LISP will understand, we will be writing the $\Re$-image, rather than the LISP expression form. Thus to ask a LISP machine to evaluate *car*[*(A . B)*] we present it with *(CAR (QUOTE (A . B)))*. What this means is that we are presenting our programs to the machine as data structures of the language. It would be like expressing programs in Fortran or Algol as arrays of integers; that is, the data structures of those languages. We will explore the implications of this approach to programming in later sections, but for now it should help to know that we will be making extensive use of the $\Re$-mapping.

---

[51] In LISP 1.5 *T* and *F* were used as the representations of t and f; the atoms *T* and *F* were (permanently) bound to values **T** and *NIL*.

You should go back and look at the *tgm*'s (Section 2.6) now that you know that they are evaluators for simple subsets of LISP expressions. Note that the only atoms which the great mothers recognize are *T* and *NIL*. Any other atoms elicit an error message. What do these other atoms represent? That is, of what objects are atoms the $\Re$-maps? Well, numerals are atoms and are the $\Re$-maps of numerals. We certainly could extend *tgmoaf* to handle this case. Atoms are translations of another class of LISP expressions; they are the translations of variables and function names. So one task before us is to incorporate a mechanism into our simple LISP evaluator which will handle evaluation of variables. We've already seen the necessary mechanism in Section 2.5 where we studied tables as an abstract data stucture. The other piece of LISP which did not appear in the evaluator for polynomials was conditional expressions. Before handling conditionals we wish to handle the informal intuitive discussion of tables in Section 2.5 in a more precise manner.

3.3 Symbol tables

One of the distinguishing features of computer science is its reliance on the ability to store and recover information. Any formalism which addresses itself to computer science must take this into account. In particular we must be able to handle the effect of assignment or binding, that is, the association of a value with a name in an environment. A common device used to accomplish this is the symbol table. This is the device we used informally in Section 2.5. We will review some of that discussion here.

In essence, a symbol table is simply a set of ordered pairs of objects; one of the elements of each pair is a name; the other is its value. This means that symbol tables can be characterized as relations or perhaps even as functions. This characterization is indeed viable. On page 73 we showed that a table could be constructed and maintained in a manner preserving set-ness. As an abstract operation, finding an element in a symbol table is also quite simple: given a set of ordered pairs and a variable, find a pair with first element the same as the given variable. This operation can be described as function application where the function being applied is the table and the argument is the name component. That is: $locate[x;tbl] = tbl(x)$.

This level of abstraction was a bit too spartan; maintaining such tables is computationally expensive, and we will have to give a *locate* algorithm sooner or later. The level of abstraction we envisioned looked on a symbol table as a sequence of pairs, each pair representing a variable and its corresponding value. The algorithms given in Section 2.5 for manipulating tables depended heavily on the implied sequencing of call-by-value and recursion. Since this was consistent with the explicit sequencing used in adding elements to the table, we achieved the desired effect. We found the expected bindings, even though there may have been other candidates in the tables. In the remaining sections of this chapter we will utilize more features of this interplay between representation of data and calling style of algorithm. Symbol tables are just the first manifestation of this phenomenon.

These simple symbol tables are also known as association lists or a-lists; thus *assoc* will be the name

of our function to search a symbol table. *assoc* will take two arguments: a name and a symbol table. It will examine the table from left to right, looking for the first match. We will need to designate a selector, *name*, to locate the name-component of a pair, and another selector, *value*, to retrieve the value component. If a pair is found, then that pair is returned; if no such pair is found, the result is undefined.

assoc[x;l] <= [*eq*[*name*[*first*[l]];x] → *first*[l];
t → *assoc*[x;*rest*[l]]]

Obviously, if the table is very long and the desired pair is close to the end of the table, then we may be in for a very long search. The search scheme encoded in *assoc* is called **linear search**, and is unnecessarily inefficient. However the phenomemona we wish to study here are not related to efficiency of searching methods. We will come back to symbol tables in Section to study the problems of efficient storage and retrieval of information. It will suffice now simply to think of a symbol table as represented in LISP by a list of dotted pairs, a name dotted with value. In this representation, then, *name*[x] <= *car*[x], and *value*[x] <= *cdr*[x]. For completeness, we should also specify a constructor. Though we won't need it for a while, its name is *mkent*, and will take a name and a value and return a new entry. Its representation here is *mkent*[x;y] <= *cons*[x;y].

Recall that we are representing variables as atoms; if *x*, *y*, and *z* had current values *2*, *3*, and *4*, then a symbol table describing that fact could be encoded as:

((X . 2) (Y . 3) (Z . 4)) .

For example:

assoc[*Y*; *((X . 2) (Y . 3) (Z . 4))*] = *(Y . 3)*
assoc[*U*; *((X . 2) (Y . 3) (Z . 4))*] = ⊥.

How are we to represent bindings of variables to non-numeric S-exprs? For example, how will we represent: "the current value of *x* is *A*"? We will place the dotted-pair *(X . A)* in the table. Now this representation is certainly open to question: why not add *(X .(QUOTE A))*? The latter notation is more consistent with our conception of representation espoused on page 50. That is, we map LISP expressions to S-expressions; perform the calculations on this representation, and finally reinterpret the result of this calculation as a LISP expression. The representation we have chosen for symbol tables obviates the last reinterpretation step. Now it will turn out that for our initial subsets of LISP this reinterpretation step simply would involve "stripping" the *QUOTE*s. The only "values" which a computation can return are constants; later things will become more difficult. Perhaps this representation of table entries is a poor one; we will see. In studying any existing language, or contemplating the design of any new one, we must question each detail of representation. Decisions made too early can have serious consequences [52].

---

[52] However, in defense of LISP, it must be remembered that LISP was conceived at the time that FORTRAN was believed to be a gift from God. Only later did we learn the true identity of the donor.

Before moving on we should probably take stock of our current position; in this section we have simply recreated the table-lookup mechanism we used in Section 2.5, only now we are paying a bit more attention to representation. We can locate things in a table and we have seen how calling functions can add values to a table. We have said nothing about adding function definitions to the tables. Abstractly we know how to extract the definition from the table and apply it. We must give an explicit representation of the storage of a function. This turns out to be a reasonably non-trivial problem. We have seen that it is possible to mechanize at least one scheme for evaluation of functions -- call-by-value, evaluating arguments from left to right. We have seen that it is possible to translate LISP expressions into S-exprs in such a way that we can write a LISP function which will act as an evaluator for such translations. In the process we have had to mechanize the intuitive devices we (as humans) might use to recall the definition of functions and to recall the current values of variables. It became clear that the mechanism of symbol tables could be used. To associate a variable with a value was easy. To associate a function name with its definition required some care. That is, part of the definition of a function involves the proper association of formal parameters with the body of the definition. (We actually need to be a bit more precise than this in describing recursive functions, but this is good enough for now.) The device we chose is called the **lambda notation.**

3.4 λ-notation

Recall our discussion of the problems of representation of function definitions. This discussion began on page 69 and our conclusion was that to represent a definition like $f[x;y]$ <= ξ we needed a symbol table entry with name f and a value part which contained the body of the definition, ξ, and the list of arguments, $[x;y]$, given with f. LISP uses the λ-notation to lend precision to our informal discussion of function representation.

Why add more notation to LISP? The λ-notation is a device invented by the logician Alonzo Church in conjunction with his studies in logic and foundations of mathematics. The λ-calculus is useful for a careful discussion of functions and is therefore applicable in a purified discussion of procedures in programming languages. We shall begin a detailed discussion of the λ-calculus and its relation to computer science in Section .

The notation was introduced into Computer Science by John McCarthy in the description of LISP. In the interest of precision we should note that there are actually several important distinctions between Church's λ-calculus and the notation envisioned by McCarthy. We will point out the discrepancies in Section .

We have been informally writing $f[x;y]$ <= $[x*y + y]$ as a definition of the function f. It is supposed to convey the following intent: f is the name of a function or rule; whenever f is supplied with two numeric arguments it is supposed to multiply those arguments and add the result to the second. The resulting sum is the desired answer. Since informality has tendencies toward ambiguity we would like to analyze the "<="-notation more closely. Though we say f is being defined, it is not f,

but *f*[*x*;*y*] which appears to the left of the "<="-symbol. First note that *f*[*x*;*y*] is not a function, *f* is. To see what *f*[*x*;*y*] means consider the following example. When we are asked to evaluate *car*[(*A* . *B*)] we say the value is *A*. *car*[(*A* . *B*)] is an expression to be evaluated; it is a LISP form. If *car*[(*A* . *B*)] is a form then so is *car*[*x*]; only now the value of the form depends on the current value assigned to the variable *x*. So the function is *car*; the form is *car*[*x*]. The function is *f*; *f*[*x*;*y*] is a form, as is *x***y* + *y*. Thus the current notation has a form on both sides of the "<=". We would like a notation which clearly shows what is being defined and what is given.

Also our notation has really been specifying more than just the name. The notation specifies the formal parameters (*x* and *y*) and the order in which we are to associate actual parameters in a call with the formal parameters of the definition (*x* with the first, *y* with the second). More subtly, the notation tells which variables in the function body are to be supplied values when the function is called. For example define *g*[*x*] <= [*x***y* + *y*]; then the expression *g*[*2*] specifies that *x* is to receive a value *2*, but leaves unspecified what the value of *y*, and indeed + and *, should be.

We also wish to have a notation so that function definitions can be inserted into the symbol table as "values" assigned to names. They will be parametric values, but they will be values. The λ-notation performs this task by preceding the function body with a list of variables, called **lambda variables** [53]; The resulting construct is preceded by "λ[" and followed by "]". Thus using the above example, the name *f* is exactly the same function as λ[[*x*;*y*] *x***y* + *y*]. We actually need a bit more than λ-notation to specify recursive functions in a natural manner. See Section 3.9. The λ-notation introduces nothing new as far as our intuitive binding and evaluation processes are concerned; it only makes these operations more clear.

One benefit of the λ-notation is that we need not give explicit names to functions in order to perform the evaluation. Evaluation of such anonymous functions should be within the province of LISP. To evaluate a λ-expression in LISP, bind the evaluated actual parameters to the λ-variables and evaluate the function body. For example, evaluate:

$$\lambda[[x;y]\ x^2 + y][2;3]\ .$$

We associate *x* with *2* and *y* with *3* and evaluate the expression:

$$x^2 + y.$$

Assuming arithmetic works as usual, this calculation should give *7* as value.

Or evaluate: λ[[*x*] *cdr*[*car*[*x*]]][((*A* . *B*). *C*)].

We bind *x* to the S-expression ((*A* . *B*). *C*) and evaluate the function body. The usual LISP evaluation scheme entails evaluating *car*[*x*] with the current binding of *x*; this result, (*A* . *B*), is passed to *cdr*. *cdr* performs its calculation and finally returns *B*.

---

[53] The list of variables is usually referred to as the **lambda list**.

The λ-notation can be used anywhere LISP expects to find a function, thus allowing us to write anonymous functions. For example:

$$\lambda[[x]\ first[x]][\lambda[[y]\ rest[y]][(A\ B)]]$$

This expression is a complicated way of writing:

$$f[g[(A\ B)]] \text{ where } f[x] <= first[x] \text{ and } g[y] <= rest[y].$$

Though the second form is perhaps easier for us to comprehend, the first form is equivalent and will be acceptable to the evaluator we will write. Indeed the mechanical evaluation of the second formulation will pass through the first on its way to complete evaluation. At any rate:

$$\lambda[[x]\ first[x]][\lambda[[y]\ rest[y]][(A\ B)]] = \lambda[[x]\ first[x]][(B)] = B$$

Still, evaluation requires care. For example, is $\lambda[[x]2]$ the constant function which always gives value *2*? It isn't in LISP. The evaluation of an expression involving this function requires the evaluation of the actual parameter associated with x. That computation may not terminate. For example, consider $\lambda[[x]2][fact[-1]]$ where *fact* is the LISP implementation of the factorial function given on page 41.

Since we intend to include λ-expressions in our language we must include an $\mathfrak{R}$-mapping of them into S-expression form:

$$\mathfrak{R}[\![\lambda[[x_1; ...; x_n]\xi]\ ]\!] = (LAMBDA\ (X_1\ ...\ X_n)\ \mathfrak{R}[\![\ \xi\ ]\!])$$

Thus the character λ will be translated to *LAMBDA*. *LAMBDA* is therefore a kind of prefix operator (like *COND*) which will help the evaluator recognize the occurrence of a function definition (just as *COND* will be used by the evaluator to recognize the occurrence of a translation of a conditional expression).

Here are some examples of λ-expressions and their $\mathfrak{R}$-translations:

$$\mathfrak{R}[\![\lambda[[x;y]\ x^2 + y]\ ]\!] = (LAMBDA\ (X\ Y)\ (PLUS\ (EXPT\ X\ 2)\ Y))$$
$$\mathfrak{R}[\![\lambda[[x;y]\ cons[car[x];y]]\ ]\!] = (LAMBDA\ (X\ Y)\ (CONS\ (CAR\ X)\ Y))$$

To make our discussion of λ-expressions completely legitimate, our LISP syntax equations should now be augmented to include:

<function> ::= λ[<varlist><form>]

<varlist> ::= [<variable>; ... ; <variable>]
::= [ ]

Besides giving a clear notation for function definitions, the λ-notation is a useful computational device.

Consider the following sketch of a function definition:

$$g <= \lambda[[x][\pi[lic[x]] \rightarrow lic[x]; .... x ...]],$$

where *lic* may be a long involved calculation. We certainly must compute *lic*[*x*] once. But as *g* is defined, we would compute *lic*[*x*] twice if p_1 is true: once in the calculation of p_1, and once as e_1. Since both calculations of *lic*[*x*] will give the same value [54], this second calculation is unnecessary. *g* is inefficient. We could write:

$$g <= \lambda[[x]f[lic[x];x]]$$

where:

$$f <= \lambda[[u;v][\pi[u] \rightarrow u; .... v ...]].$$

In this scheme *lic* will only be evaluated once; its value will be passed into *f*. Using λ-expressions, in a style called **internal lambdas** we can improve *g* without adding any new function names to our symbol tables as follows:

Replace the body of *g* with:

LAM $\lambda[[y][\pi[y] \rightarrow y; ... x ...]][lic[x]].$

Call this new function *g'*:

```
g' <= λ[[x]
        λ[[y][π[y] → y; ... x ... ]][lic[x]]
        ].
```

Now when *g'* is called we evaluate the actual parameter, binding it to *x*, as always; and evaluate the body, LAM. Evaluation of LAM involves calculation of *lic*[*x*] once, binding the result to *y*. We then evaluate the body of the conditional expression as before. If p_1 is true, then this definition of *g'* involves one calculation of *lic*[*x*] and two table look-ups (for the value of *y*), rather than the two calculations of *lic*[*x*] in *g*. More conventional programming languages can obtain the same effect as this use of internal lambdas by assignment of *lic*[*x*] to a temporary variable. We will introduce assignment statements in LISP in Section 3.13.

---

[54] Our current LISP subset has no side effects. That means there is no way for a computation to affect its surrounding environment. The most common construct which has a side-effect is the assignment statement.

Problems

I. What is the difference between $\lambda[[\ ]\ x*y + y]$ and $x*y + y$?

3.5 Mechanization of evaluation

We now have picked a representation for LISP expressions; have introduced a precise notation for discussing functions; and we have given plausibility arguments for the existence of an evaluator for LISP. It is now time to write a formal, precise, evaluator for LISP expressions. The evaluator will be the final arbiter on the question of the meaning of a LISP construct. The evaluator is thus a very important algorithm. We will express it and its related functions in as representation-free form as possible, but we will always have our Cambridge polish representation in the back of our minds.

As we have said, *tgmoaf* and *tgmoafr* (Section 2.6) are evaluators for subsets of LISP. Armed with our symbol-table mechanism we could now extend the great-mothers to handle variable look-ups. Rather than do this we will display our burgeoning cleverness and make a total revision of the structure of the evaluators. In making the revision, the following points should be remembered:

1. Expressions to be evaluated can contain variables, both simple variables and variables naming λ-expressions. Thus evaluation must be done with respect to an environment or symbol table. We wish to recognize other (translations of) function names besides *CAR*, *CDR*, *CONS*, *EQ*, and *ATOM* in our evaluator, but explicitly adding new definitions to the evaluator in the style of the recognizers for the five primitives is a bad idea. Our symbol table should hold the function definitions and the evaluator should contain the general schemes for finding the definitions, binding variables to values, and evaluating the function body. By now you should be convinced that this process is a reasonably well defined task and could be written in LISP.

2. All **function** calls are to be evaluated by-value. However, there are some **special forms** we have seen which are not evaluated in the normal manner. In particular, conditional expressions, quoted expressions, and lambda expressions are handled differently.

The primary function is named *eval* rather than *sogm* (Son of Great Mother). It will take two arguments; the first is a representation of an expression to be evaluated, and the second is a representation of a symbol table. The function *eval* will recognize numbers, and the constants *T* and *NIL*, and if presented with a variable, will attempt to find the value of the variable in the symbol table using *assoc* (Section 3.3).

eval also handles the special forms *COND* and *QUOTE*. If *eval* sees a conditional expression (represented by *(COND ...)*) then the body of the *COND* is passed to a subfunction named *evcond*. *evcond* embodies the conditional expression semantics as described on page 18. The special form, *(QUOTE α)*, signifies the occurrence of a constant, α, which is simply returned. As far as this *eval*

is concerned, any other expression is a call-by-value function application. The argument-list evaluation is handled by *evlis* in the authorized left-to-right ordering. This is handled by recursion on the sequence of arguments. In this case we apply the function to the list of evaluated arguments. This is done by the function *apply*.

With this short introduction we will now write a more general evaluator which will handle a larger subset of LISP than the *tgms*. Here's the new *eval*:

```
eval <= λ[[exp;environ]
        [isvar[exp] → value[assoc[exp;environ]];
         isconst[exp] → denote[exp];
         iscond[exp] → evcond[rest[exp];environ];
         isfunc+args[exp] → apply[func[exp];evlis[arglist[exp];environ];environ]]
```

and:

```
denote <= λ[[exp]
        [isnumber[exp] → exp;
         istruth[exp] → exp;
         isfalse[exp] → exp;
         isSexpr[exp] → rep[exp]
        ]]
```

where:

rep knows how to extract the S-expr from the representation. In our scheme the selector *rep* is given by *cadr*.

```
evcond <= λ[[e;environ]
        [eval[ante[first[e]];environ] → eval[conseq[first[e]];environ];
         t → evcond[rest[e];environ]]]
```

and,

```
evlis <= λ[[e;environ]
        [null[e] → ( );
         t → concat[eval[first[e];environ];evlis[rest[e];environ]]]]
```

The selectors, constructors and recognizers which relate this abstract definition to our particular S-expression representation are grouped with *apply* on page 95. The subfunctions, *evcond* and *evlis*, are simple. *evcond* you've seen before in *tgmoafr* in a less abstract form; *evlis* simply manufactures a new list consisting of the results of evaluating (from left to right) the elements of *e*, using the symbol table, *environ*, where necessary. Since *evcond* and *evlis* are LISP functions, they are subject to the left-to-right evaluation rule. Thus *evlis* embodies the left-to-right rule. If *evlis* were evaluated under a right-to-left rule then *evlis* would evaluate expressions in right-to-left order. It is possible to write a version of *evlis* which only depends on being evaluated CBV, and which does embody the left-to-right rule:

```
evlis <= λ[[e;environ]
        [null[e] → ( );
         t → λ[[x]concat[x;evlis[rest[e];environ]]]
              [eval[first[e];environ]]
        ]]
```

Another application of the left-to-right property occurs within *apply* in the symbol table search and construction process. We know that *assoc* looks from left to right for the latest binding of a variable. Thus the function which augments the table must add the latest binding to the front. When do new bindings occur? They occur at λ-binding time, and at that time the function *pairlis* will build an augmented symbol table with the λ-variables bound to their evaluated arguments.

The function *apply* takes three arguments: a representation of a function, a representation of a list of evaluated arguments, and a representation of a symbol table. *apply* explicitly recognizes the representations of the five primitive functions *CAR*, *CDR*, *CONS*, *EQ*, and *ATOM*. If the function name is a variable, the definition is located in the symbol table by *eval* and applied to the arguments. Otherwise the function must be a λ-expression. This is where things get interesting. We know we must evaluate the body of the λ-expression after binding the formal parameters of the λ-expression to the evaluated arguments. How do we bind? We add variable-value pairs to the symbol table. We will define a subfunction, *pairlis*, to perform the binding. Then all that is left to do is give the function body and the new symbol table to *eval*. Here is *apply*:

```
apply <= λ[[fn;args,environ]
                [iscar[fn] → car[arg₁[args]];
                 iscons[fn] → cons[arg₁[args];arg₂[args]];
                     ...          ...
                 isvar[fn] → apply[eval[fn;environ];args;environ];
                 islambda[fn] → eval[body[fn];pairlis[vars[fn];args;environ]]
                 ]]

pairlis <= λ[[vars;vals;environ]
                [null[vars] → environ;
                 t → concat[mkent[first[vars];first[vals]];pairlis[rest[vars];rest[vals];environ]]
                 ]]
```

Some of the functions and predicates which will relate these abstract definitions to our specific S-expression representation of LISP constructs are:

Recognizer	Selector	Constructor
iscar <= λ[[x]eq[x;CAR]]	*func* <= λ[[x]first[x]]	*mkent* <= λ[[x;y]cons[x;y]]
isSexpr <= λ[[x]eq[first[x];QUOTE]]	*arglist* <= λ[[x]rest[x]]	
istruth <= λ[[x] eq[x;T]]	*body* <= λ[[x]third[x]]	
	vars <= λ[[x]second[x]]	

The only really new development is in the λ-binding process. Notice that *pairlis* makes a new symbol table by consing new pairs onto the front of the old symbol table; and recall that *assoc* finds the first matching pair in the symbol table. This is important.

To summarize then: the evaluation of an expression $f[a_1; ... ;a_n]$, where the a_i's are S-exprs, is the same as the result of applying *eval* to the $\Re$-translation, $(\Re[\![f]\!], \Re[\![a_1]\!], ... \Re[\![a_n]\!])$. This behavior is again an example of the diagrams of page 50. In its most simple terms, we mapped LISP evaluation onto the LISP *eval* function; mapped LISP expressions onto S-expressions; and executed *eval*. Notice that in this case we do not reinterpret the output since the structure of the representation does this implicitly. We have commented on the efficacy of this already on page 88.

This specification of the semantics of LISP using *eval* and *apply* is one of the most interesting developments of computer science.

Problems

I. Compare our version of *eval* and *apply* with the version given in the appendix. Though the current version is much more readable, how much of it still depends on the representation we chose? That is, how abstract is it really?

II. Complete the specification of the selectors, constructors, and recognizers.

3.6 Examples of *eval*

We will demonstrate the inner workings of the evaluation algorithm on a couple of samples and will describe the flow of control in the execution in a couple of different ways. The examples will be done in terms of the image of the $\Re$-mapping rather than being done abstractly. We do this since the structure of an actual LISP evaluator will use this representation [55]. It is important that you diligently study the sequence of events in the execution of the evaluator. The process is detailed and tedious but it must be done once so that you may convince yourself of its correctness.

Let's evaluate $f[2;3]$ where f <= $\lambda[[x;y]\ x^2 + y]$. That is, evaluate:

$$eval[\ \Re[\![\ f[2;3]\ ]\!];\ \Re[\![\{\ <f,\ \lambda[[x;y]\ +[\uparrow[x;2];\ y]]>\ \}]\!]].$$

After appropriate translation this is equivalent to evaluating:

eval[(F 2 3); ((F . (LAMBDA (X Y) (PLUS (EXPT X 2) Y))))]

[55] Recall that we will be programming in the $\Re$-image.

Notes:

1. *((F . (LAMBDA (X Y) ...)))* = *((F LAMBDA (X Y) ...))* This is mentioned because LISP implementations will print the latter even if you write the former.

2. Since the symbol table *((F ...))* occurs so frequently in the following trace, we will abbreviate it as *st*. Note that we have no mechanism yet for permanently increasing the repertoire of known functions. We must resort to the subterfuge of initializing the symbol table to get the function f defined.

3. For this example we must assume that + and ↑ (exponentiation) are known functions. Thus *apply* would have to contain recognizers for *PLUS* and *TIMES*:

... $atom[fn] \rightarrow [isplus[fn] \rightarrow +[arg_1[args];arg_2[args]];$
$isexpt[fn] \rightarrow \uparrow[arg_1[args];arg_2[args]];$
...]

...

So *eval*[(*F* 2 3);*st*]

= *apply*[*func*[(*F* 2 3)]; *evlis*[*arglist*[(*F* 2 3)];*st*];*st*]
= *apply*[*F*;*evlis*[(2 3);*st*];*st*]
= *apply*[*F*;(2 3);*st*]
= *apply*[*eval*[*F*;*st*];(2 3);*st*]
= *apply*[(*LAMBDA* (*X Y*) (*PLUS* (*EXPT X* 2) *Y*)); (2 3);*st*]

= *eval*[*body*[(*LAMBDA* (*X Y*) (*PLUS* (*EXPT X* 2) *Y*))];
pairlis[*vars*[(*LAMBDA* (*X Y*) (*PLUS* (*EXPT X* 2) *Y*))];(2 3);*st*]]

= *eval*[(*PLUS* (*EXPT X* 2) *Y*);*pairlis*[(*X Y*);(2 3);*st*]]
= *eval*[(*PLUS* (*EXPT X* 2) *Y*);((*X* . 2)(*Y* . 3)(*F LAMBDA* (*X Y*) ...))]

= *apply* [*PLUS*; *evlis*[((*EXPT X* 2) *Y*);((*X* . 2)(*Y* . 3)..)];((*X* . 2)...)]

Let's do a little of: *evlis*[((*EXPT X* 2) *Y*);((*X* . 2)(*Y* . 3)...)]
= *concat*[*eval*[(*EXPT X* 2);((*X* . 2)(*Y* . 3) ...)];
evlis[(*Y*);((*X* . 2) ...)]]
= *concat*[*apply*[*EXPT*;*evlis*[(*X* 2);((*X* . 2)...)];((*X* . 2) ...]
evlis[(*Y*); ...]]

= *concat*[*apply*[*EXPT*;(2 2);((*X* . 2) ...];*evlis*[(*Y*); ...]]
= *concat*[↑[arg_1[(2 2)];arg_2[(2 2)]];*evlis*[(*Y*); ...]]
= *concat*[↑[2;2];*evlis*[(*Y*); ...]]
= *concat*[4;*evlis*[(*Y*);((*X* . 2)(*Y* . 3) ....)]]
= *concat*[4;*concat*[*eval*[*Y*;((*X* .2) ...)]; *evlis*[( );((...))]]]
= *concat*[4;*concat*[3;( )]]
= (4 3)

Now back to apply:

= *apply*[*PLUS*;(4 3);((*X* . 2)(*Y* . 3) ...)]
= +[4;3]
= 7

It should now be clear that *eval* and friends do perform as you would expect. It perhaps is not clear that a simpler scheme might not do as well. In particular, the complexity of the symbol table mechanism which we claimed was so important has not been exploited. The next example will indeed show that a scheme like ours is necessary to keep track of variable bindings.

Let's sketch the evaluation of *fact*[3] where:

fact <= λ[[*x*][*x* = 0 → 1;t → ∗[*x*;*fact*[*x*-1]]]];

that is, *eval*[(*FACT* 3);*st*] where *st* names the initial symbol table:

((*FACT* .(*LAMBDA* (*X*) (*COND* ((*ZEROP X*) 1) (*T* (*TIMES X* (*FACT* (*SUB1 X*)))))))).

In this example we will assume that the binary function $*$, the unary predicate *zerop* <= λ[[x]x = *0*] and unary function *sub1* <= λ[[x] x-*1*] are known and are recognized in the evaluator as *TIMES*, *ZEROP* and *SUB1* respectively.

Then *eval*[(*FACT 3*);*st*]
 = *apply*[*FACT*;*evlis*[(*3*);*st*];*st*]
 = *apply*[(*LAMBDA* (*X*) (*COND* ...));(*3*);*st*]
 = *eval*[(*COND* ((*ZEROP X*) *1*) (*T* (...)));((*X* . *3*) . *st*)]
 = *evcond*[(((*ZEROP X*) *1*) (*T* (*TIMES X* (*FACT* (*SUB1 X*)))));((*X* . *3*) . *st*)]
(Now, let *st1* be ((*X* . *3*) . *st*))
 = *eval*[(*TIMES X* (*FACT* (*SUB1 X*))); *st1*]
 = *apply*[*TIMES*;*evlis*[(*X* (*FACT* (*SUB1 X*))); *st1*];*st1*]
 = *apply*[*TIMES*;*concat*[*3*;*evlis*[((*FACT* (*SUB1 X*))); *st1*]];*st1*]

Now things get a little interesting inside *evlis:*
evlis[((*FACT* (*SUB1 X*)));*st1*]
 = *concat*[*eval*[(*FACT* (*SUB1 X*)); *st1*];( )]
 and *eval*[(*FACT* (*SUB1 X*));*st1*]
 = *apply*[*FACT*;*evlis*[((*SUB1 X*));*st1*];*st1*]
 = *apply*[*FACT*; (2);*st1*]
 = *apply*[(*LAMBDA* (*X*) (*COND* ...));(2);*st1*]
 = *eval*[(*COND* ((*ZEROP X*) *1*) ...));((*X* . 2) . *st1*)]
 ...

Notice that within this latest call on *eval* the symbol-table-searching function, *assoc*, will find the pair (*X* . *2*) when looking for the value of *x*. This is as it should be. But notice also that the older binding, (*X* . *3*), is still around in the symbol table *st1*, and will become accessible once we complete this latest call on *eval*. It will become accessible because this earlier manifestation of the table was saved by the λ-binding process as we entered the inner call on *eval*; as we leave this inner evaluation, the previous incarnation of the table is restored.

As the computation continues, the current symbol table appears as follows:

((*FACT LAMBDA* (*X*) (*COND* ...))) = *st*,
((*X* . *3*) . *st*) = *st*′,
((*X* . 2) . *st*′) = *st*′′,
((*X* . *1*) . *st*′′) = *st*′′′,
((*X* . *0*) . *st*′′′).

Thus each new level of the table builds on the prior *st*; each prior *st* is saved in the following line from *apply* (page 95):

islambda[*fn*] → *eval*[*body*[*fn*];*pairlis*[*vars*[*fn*];*args*;*environ*].

The call on *eval* is performed with the augmented table; when we leave that inner *eval* we return to an environment which contains the prior *st*.

We claim that using *pairlis* to concat the new bindings onto the front of the symbol table as we call *eval* does the right thing. The tricky part is that when we leave that particular call on *eval*, the old table is automatically restored by the recursion mechanism. That is, if *st* is the current symbol table then *concat*-ing things onto the front of *st* doesn't change *st*, but if we call *eval* or *apply* with a symbol table of say:

$$concat[(X\ .\ 2);concat[(X\ .\ 3);\ st]]$$

then in that call on *eval* or *apply* we have access to $x = 2$, not $x = 3$.

Since the search function, *assoc*, always proceeds from left to right through the table and since the table entry function, *pairlis*, always *concats* pairs onto the left of the table before *eval* is called, we will get the correct binding of the variables.

This symbol table mechanism is very important, so let's look at it again in a slightly different manner.

In this example, expressions and table entries will be written more informally. Since the evaluator is operating on the S-expr representation of expressions we should continue to present these arguments to *eval* as S-exprs. However, the object being represented is frequently more understandable and readable than the representation of that object. Thus, initially, we will write *quote*[ξ] [56] rather than the explicit $\mathfrak{R}$-image of ξ; for example, write *quote*[*fact*[*3*]] rather than *(FACT 3)*. Later we will simply write ξ, without the *quote* where no confusion is likely. With similar motivation, we represent the symbol table between vertical bars, "|", in such a way that if a table, t_1, is:

```
| b_n   |
|  ...  | then concating a new element, b_{n+1} onto t_1 gives:
| b_1   |
```

```
| b_{n+1} |
| b_n     |
|  ...    |
| b_1     |
```

Note that the elements of the table should also be presented as S-exprs. We will represent the entires in a more transparent form as simple equations. Thus, for example:

$eval[quote[fact[3]];\ |\ fact = \lambda[[x][x=0 \to 1;t \to *[x;fact[x-1]]]]\ |\ ]$

---

[56] See page 86 for *quote*.

```
= eval[quote[[x=0 → 1; t → *[x;fact[x-1]]]];      |x = 3           |
                                                  |fact = λ[[ ...  |  ]

= *[3;eval[quote[[x=0 → ...];                     |  x = 2         |
                                                  |  x = 3         |
                                                  |fact = λ[[ ...  |  ]

= *[3; *[2;eval[quote[[x=0 → ...];                |  x = 1         |
                                                  |  x = 2         |
                                                  |  x = 3         |
                                                  |fact = λ[[ ...  |  ]

= *[3; *[2; *[1;eval[quote[[x=0 → ...];           |  x = 0         |
                                                  |  x = 1         |
                                                  |  x = 2         |
                                                  |  x = 3         |
                                                  |fact = λ[[ ...  |  ]

= *[3; *[2; *[1;1]]]  with:                       |  x = 1         |
                                                  |  x = 2         |
                                                  |   ...          |

= *[3; *[2;1]]        with:                       |  x = 2         |
                                                  |   ...          |

= *[3;2]              with:                       |  x = 3         |
                                                  |   ...          |

= 6.                  with:                       |fact = λ[[ ... |.
```

= 6

Notice that after we went to all the trouble to save the old values of x we never had to use them. However, in the general case of recursive evaluation we must be able to save and restore the old values of variables.

For example, recall the definition of *equal:*

```
equal <= λ[[x;y]
            [atom[x] → [atom[y] → eq[x;y];t → f];
             atom[y] → f;
             equal[car[x];car[y]] → equal[cdr[x];cdr[y]];
             t → f]]
```

If we were evaluating:

equal[((A . B) . C);((A . B) . D)],

then our symbol table structure would change as follows:

```
|equal = λ[[x;y] ...| ==>            |x = ((A . B) . C)| ==>
                                     |y = ((A . B) . D)|
                                     |equal = λ[[x;y]...|

| x = (A . B)     |                  |     x = A        |
| y = (A . B)     |                  |     y = A        |
| x = ((A . B). C) | ==>             |  x = (A . B)     |
| y = ((A . B). D) |                 |  y = (A . B)     | ==>
|equal = λ[[x;y ... |                | x = ((A . B). C) |
                                     | y = ((A . B). D) |
                                     |equal = λ[[x;y] ...|

|      x = B       |                 |     x = C        |
|      y = B       |                 |     y = D        |
|   x = (A . B)    |                 | x = ((A . B). C) | ==>
|   y = (A . B)    | ==>             | y = ((A . B). D) |
| x = ((A . B). C) |                 |equal = λ[[x;y] ... |
| y = ((A . B). D) |
|equal = λ[[x;y] ... |

                    |equal = λ[[x;y] ... |
```

This complexity is necessary, for while we are evaluating *equal*[*car*[*x*];*car*[*y*]], we rebind *x* and *y* but we must save the old values of *x* and *y* for the possible evaluation of *equal*[*cdr*[*x*];*cdr*[*y*]].

Before moving on we should examine *eval* and *apply* to see how they compare with our previous discussions of LISP evaluation. The spirit of call-by-value and conditional expression evaluation is maintained. λ-binding seems correct, though our current discussion is not complete. At least one preconception is not maintained here. Recall the discussion on page 16. We wanted n-ary functions called with exactly n arguments. An examination of the structure of *eval* and *apply* shows that if a function expecting n arguments is presented with fewer, then the result is undefined; but if it is given more arguments than necessary then the calculation is performed. For example:

eval[(CONS(QUOTE A)(QUOTE B)(QUOTE C));NIL]
= *eval*[(CONS(QUOTE A)(QUOTE B));NIL]
= (A . B)

This shows one of the pitfalls in defining a language by an evaluator. If the intuitions of the language specifiers are faulty or incomplete then either we are faced with maintaining that faulty judgement, or we must lobby for a "revised report".

The definition of a language by an evaluator written in that language is subject to other criticisms. The troublesome areas of our description of LISP's evaluation included λ-binding, calling styles in general and call-by-value in particular, and left-to-right order of evaluation. We wrote *eval* to explicate the meaning of these constructs, yet within *eval* we often relied on exactly these constructs to convey our intent. Now, our description in not entirely circular; *eval* does convey much of our intention to the reader. However much of the discussion of how these constructs operate is either implicit or is explained by using the same constructs. In gaining a clearer understanding of what LISP constructs mean, *eval* is exemplary. Indeed many of the details of how these constructs work are irrelevant to such an understanding. However, when we attempt to implement a language feature we cannot assume the existence of that feature; the implementation must be prepared from a combination of more primitive components. As we procede through the text we will make explicit the mechanisms which are necessary to correctly implement LISP's constructs and indeed the constucts of most other languages. In Section we give several alternative algorithms for *eval*. They will evolve to an *eval* which makes explicit most of the mechanisms we need. In Section we will begin to discuss efficient representations for LISP's data structures, control structures, and primitive operations.

The remainder of this chapter will explicate further features of LISP in preparation for that discussion.

Problems

I Which parts of *eval* and Co. allow correct evaluation of functions applied to too many arguments?

II On page 102 we noted that the evaluator performs "correctly" when evaluating forms like *cons*[*A*;*B*;*C*]. Can you find other anomalies in *eval* and friends? That is, places where unexpected results are obtained?

3.1 Variables

Let's look more closely at λ-binding in *eval*. The scheme presented seems reasonable, but as in the case *cons*[*A*;*B*;*C*], there may be more going on here than we are perhaps aware of.

If we asked *eval* to compute *f*[2], given a representation for *f* <= λ[[*x*] *x* + *y*] but no represenation for the value of *y* it would complain. It would it would find *f*, bind *2* to *x*; it would begin the evaluation of the body of *f*, finding *x*'s value, but then would find no value for *y*. However, if we asked it to evaluate the form λ[[*y*]*f*[*2*]][*1*] it would work. It would find the value of *y* to be *1* and would get a final answer of *3* as expected. You should convince yourself of this assertion.

Within the evaluation of *f*[2] in λ[[*y*]*f*[2]][*1*] the variable *y* has a different character from that of *x*. The value of *x* is found within the latest λ-binding, whereas *y* was bound in a dynamically surrounding λ-binding. That is, the λ-expression which bound *y* took effect before the binding of *x* and is still in effect when the binding of *x* is made. We do have access to *y*'s binding in this case; the *assoc* routine will locate *y*'s value. There is a third kind of name-value association present in these examples: we expect that the symbol "+" is recognized during the evaluation as denoting a procedure for computing the sum of two numbers. In previous discussions we have assumed that "+" was pre-defined inside *apply* and therefore explicitly recognized. Finally, in the first example, a fourth kind of variable usage occurred. The variable *y* had no associated value when the computation expected one. In this section we wish to examine properties of variables.

The implementation of λ-bindings described in *evlis* (page 95) is slightly misleading. There, the new λ-bindings are *concat*-ed onto the front of the existing table. They go on in a one-at-a-time fashion even though they are to be thought of as a logical unit: at the language level they all go on together, and they all come off together. It is the structure of this table which we should also examine. To these ends we now introduce some terminology.

Consider the evaluation of the expression:

λ[[*y*]*equal*[λ[[*x*]*cons*[*x*;*y*]][(*A* . *B*)];*x*]][*A*]

in an environment where the definition of *equal* is known.

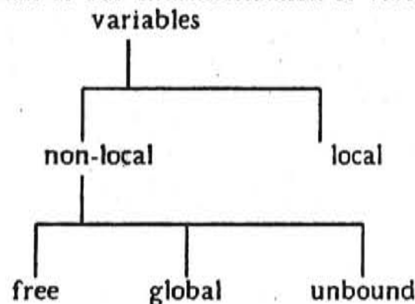
We evaluate the main argument *A*, and perform the λ-binding of *A* to *y*. This operation of λ-binding creates what we call a **local symbol table** and the variables bound in that local table are called **local bindings** for the body of the λ-expression. We now begin the evaluation of the arguments to *equal*. The first argument is itself an expression requiring λ-binding. We evaluate it's argument and bind (*A* . *B*) to *x*. This creates a local binding for *x*. In the process of making *x* local what happens to *y*? Notice that the binding process has not made *y* inaccessible: we can compute *cons*[*x*;*y*] even though *y* is not local. Variables like *y* which are accessible, but not local, we call **non-local** variables. Thus both *y* and *cons* are non-local variables in our evaluation of *cons*[*x*;*y*]. There is a further distinction between *y* and *cons*: We expect *cons* to be a predefined function; indeed *cons* has not been λ-bound any where in our computation. Variables like *cons* we will call **global variables.**

Global variables include predefined function names, *car*, *cdr*, etc, and variables like *t* and *nil*. A useful interpretation of global variables is that they are bound in the initial symbol table, also called the **global table** [1]. Non-local variables which are λ-bound somewhere in the symbol table we call **free variables**, and variables which have some accessible binding at the current point in the computation are called **bound variables** [2].

Finally the first argument to *equal* is evaluated giving *((A . B) . A)*. As we complete that evaluation the local binding for *x* is removed, and *y* becomes local again. We examine the second argument to *equal*, *x*, and find there is now no binding for that variable. Variables which have no binding of any kind at the time we ask for a value are called **unbound variables.** The local, free, and global variables make up the class of bound variables.

For a computation to be meaningful, each variable which that computation references must be bound **somewhere** when we ask for its value. So the computation of our current example would fail. Since we are doing call-by-value, it would fail even before we asked for the definition of *equal*. One of our tasks will be to discuss where definitions such as that for *equal* should be kept.

Here is a diagram of our characterization of variables:



Notice that a variable which is initially global may become local and then non-local by virtue of λ-bindings.

One of the difficulties in programming languages is deciding what value to associate with a non-local variable. In LISP, it is clear how values get associated; it happens through λ-binding or by virtue of an initial entry in the symbol table. The binding strategy for local variables is reasonably uniform in programming languages: bind some form of the actual parameter [3] to the

---

[1] This analogy breaks down somewhat in that usual implementations of LISP allow this global table to be augmented; for example, by function definitions using a version of "<=". Thus the global table can be enlarged whereas a true λ-binding involves a fixed number of variables.

[2] Our notion of free and bound variables has a decidedly computational flavor, in contrast to the mathematical definitions of "free" and "bound".

[3] evaluated or unevaluated

formal parameter and evaluate the body of the definition. The difficulty is that frequently there will be choices of values to associate. The scheme which LISP uses for discovering the value of any variable is to proceed linearly down the symbol table, looking for the **latest** binding. This scheme is called **dynamic binding**. It usually results in uncovering the value that is expected; but not always. Conceptually, the dynamic binding scheme corresponds to the physical replacement of the function call with the function body and then an evaluation of the resulting expression.

In review, the evaluation of a typical function-call will involve the evaluation of the arguments, the binding of the λ-variables to those values, the addition of these new bindings to the front of the symbol table, and finally the evaluation of the body of the function. That segment of the symbol table which we have just added by the λ-binding will be called the **local symbol table** or local environment. The variables which appear in that segment are the local variables. The remainder of the symbol table makes up the **non-local table**. Variables which appear in the global table but not in any local table are the **global variables**. Free variables are bound somewhere between the local table and the global table. Variables which are local to a form-evaluation are those which were present in the λ-binding. We first wish to develop a useful notation for describing bindings before delving further into the intricacies of binding strategies. That discussion will be the content of Section 3.11.

Problems

I Write a LISP predicate, *non* <= λ[[*x*;*e*] ...], which will give t just in the case that *x* and *e* represent a variable and a λ-expression respectively, and *x* is non-local to *e*.

3.8 Environments and bindings

This section will introduce one more notation for describing symbol tables or environments. This notation, due to J. Weizenbaum, only shows the abstract structure of the symbol table manipulations during evaluation. Its simplicity will be of great benefit when we introduce the more complex binding schemes necessary for function-valued functions in Section 3.10.

In the previous discussions it has been sufficient simply to think of a symbol table as a sequence of pairs; each pair was a variable and its associated value. This sufficed because we dealt only with λ-variables; we ignored the possibility of free variables. As long as we added the λ-bindings to the **front** of the sequence representing the symbol table we showed that expected evaluation would result. Local values were found in the table; global values were found by explicit recognizers in *eval* and *apply*. With the advent of free variables, however it is convenient, and soon will be necessary, to examine the structure of environments more closely. We will describe our environments in terms of a local symbol table augmented by a description of where to look for the non-local values.

Instead of having one amorphous sequential symbol table, we envision a sequence of tables. One is the local table, and its successor in the sequence is the previous local table. The information telling

where to find the previous table is called the access chain or access link. Thus if tables are represented by E_i and the access link by → then we might represent a symbol table as:

$$(E_n \rightarrow E_{n-1} \rightarrow \dots \rightarrow E_1 \rightarrow E_0)$$

where E_n is the local or current segment of the table. We reserve E_0 to name the global table.

LISP thus finds local bindings in the local table and uses the access chain to find bindings of non-local variables. If a variable is not found in any of the tables, then it is unbound.

Now to establish some notation. An environment will be described as :

Form
E_{local}
| E_i

var	value
v_1	val_1
v_2	val_2
.....	
v_n	val_n

Form is the current form being evaluated. E_{local} is the name of the current environment or symbol table. Let x be a variable appearing in Form. If x is not found among the v_j's, then entries in the table named E_i are examined. If the variable is not found in E_i then the environment mentioned in the upper right-hand quadrant of E_i is searched. The search will terminate if the variable is found; the value is then the corresponding val_j. If x is not found locally, and the symbol "/" appears in the right-hand quadrant, then x is unbound.

The notation is used as follows: when we begin the evaluation of a form, the initial table E_0 is set up with "/" in its access field. The execution of a function definition, say $f \Leftarrow \lambda[[x;y]x^2 + y]$, will add an appropriate entry to the table, binding f to its lambda definition [4]. Now, consider the evaluation of the form $f[2;3]$. When the λ-expression is entered, i.e., when we bind the evaluated arguments (2 and 3) to the λ-variables (x and y), a new local table (E_1) is set up with an access link to E_0. Entries reflecting the binding of the λ-variables are made in E_1 and evaluation of the λ-body is begun.

[4] Note that we really mean "representation of lambda definition". However the informal notation is easier to read and thus we will continue the deception.

The flow of symbol table creation is:

```
        f[2;3]                    x² + y
        E0                        E1                          E0
        |/                        | E0                        |/
        _______ =>                _______        =>           _______
f  | λ[[x;y]x² + y]           x | 2                   f | λ[[x;y] ...
                              y | 3
```

Compare this sequence to the example on page 98.

The sequence of tables corresponds to the evaluation sequence:

$$eval[\Re[\![f[2;3]]\!];\ \Re[\![\{\langle f, \lambda[[x;y]\ x^2 + y]\rangle\}]\!]]$$

$$\downarrow$$

$$eval[\Re[\![x^2 + y]\!];\ \Re[\![\{\langle x, 2\rangle, \langle y, 3\rangle, \langle f, \lambda[[x;y]\ x^2 + y]\rangle\}]\!]]$$

$$\downarrow$$

$$7$$

The execution of *fact*[*3*] on page 98 results in a more interesting example. The following discussion should be read in conjunction with that description [61].

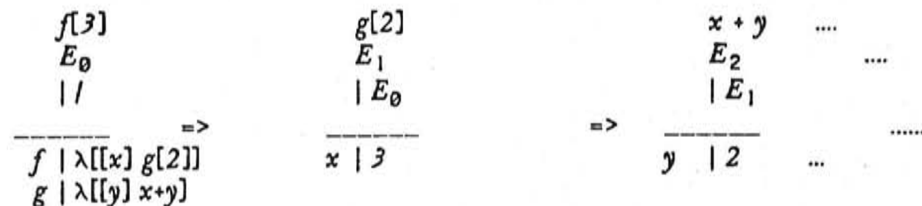
```
                                                                      |-> 6
        fact[3]              [x=0-> ...]              *[x;fact[x-1]]  |<-
        E0                   E1                       E1              | |
        | /                  | E0                     | E0            ┘ ↑
       ______       =>      ______           =>      ______ =>          2
fact   | λ[[x][x=0->1;...]  x   | 3                  x   | 3             ↑

        fact[2]              [x=0-> ...]              *[x;fact[x-1]]
        E1                   E2                       E2              |<-
        | E0                 | E1                     | E1            ┘ ↑
=>     ______       =>      ______           =>      ______ =>          1
     x  | 3                 x   | 2                  x   | 2             ↑

        fact[1]              [x=0-> ...]              *[x;fact[x-1]]
        E2                   E3                       E3              |<-
        | E1                 | E2                     | E2            ┘ |
=>     ______       =>      ______           =>      ______ =>          |
     x  | 2                 x   | 1                  x   | 1             ↑
                                                                         1
                                                                         ↑
        fact[0]              [x=0->1; ...]
        E3                   E4                                       | ↑
        | E2                 | E3                     send            | |
=>     ------       =>      ------           => 1     1               |->
     x  | 1                 x   | 0                   back up         ┘
```

At the end of the first line we are faced with the evaluation of $*[x;fact[x-1]]$. This requires the evaluation of the arguments to $*$; this is done by *evlis* and requires the evaluation of *fact*[x-*1*] using the environment E_1. In E_1 x-*1* gives *2*, and we find the definition of *fact* in E_0. In line two we set up E_2 and evaluate *fact*[2]. Analogous situations occur until line four; at this time we suddenly find ourselves in E_4 with x bound to *0*. The predicate $x=0$ is satisfied and we start back up the right margin to conclude the nested evaluations of $*[x;fact[x-1]]$. This process finally terminates at the top, returning a value *6*.

[61] The layout of this example is due to R. Davis.

Notice in this example that we will get the correct binding of x locally. It is important to note that the occurrence of *fact* within the body of the definition of *fact* is global. We find the correct binding for *fact* by searching the access chain. We must search the access chain even though *fact* is global. We cannot shortcut the search by simply looking in E_0. A variable might have been rebound in an enclosing environment and it would be that binding we should discover.

As a final example showing access to non-local variable bindings consider *f*[3] where *f* <= λ[[*x*]*g*[2]] and *g* <= λ[[*y*]*x*+*y*].



Notice that when we attempt to evaluate $x + y$ we find y has a local value, but we must look down the access chain to find a binding for x.

The scheme for using Weizenbaum environments for the current LISP subset is:

> When doing a λ-binding, set up a new E_{new} with the λ-variables as the local variable entries and the values of the arguments as the corresponding value entries. The access slot of the new E_{new} points to the previous symbol table. The evaluation of the body of the λ-expression takes place using the new table; when a local variable is accessed we find it in E_{new}; when a non-local or global variable occurs, we chase the access chain to find its value.
>
> When the evaluation of the body is completed, E_{new} disappears and the previous environment is restored.

You should convince yourself that the current access- and binding-scheme espoused by LISP is faithfully described in these diagrams.

Problems

1. By now you should realize that environments really are a class of abstract data structures: they should have constructors, selectors, and recognizers. To help discover what a set of such functions might be, write a new version of the symbol table searching function which operates on Weizenbaum environments. The function should search the local table first, then if the variable is not found there, use the access chain information to search further. After writing the abstract version of the function give a representation for the environments and give a set of functions which you think would be useful in manipulating environments.

3.9 *label*

One effect of placing "λ" and a list of λ-variables in front of an expresson is to bind those variables which appear in the λ-list. All other variables appearing in the expression are non-local. For example, f is non-local in the following:

$$f \Leftarrow \lambda[[x][zerop[x] \rightarrow 1;\ t \rightarrow *[x;f[x\text{-}1]]]\ ].$$

Clearly our intention is that the f appearing to the right of "<=" is the same as the f appearing to the left of "<=".

This has not been a problem for us. We have simply pre-loaded the symbol table, binding f to its definition (or value); see page 98. LISP has been equipped with a more elegant device for this binding, called the *label* operator. It is used thus:

label[<identifier>;<function>]

and has the effect of binding the <identifier> to the <function>. The value constructed by executing a *label*-expression is a representation of a function with name <identifier> and body <function>. For example, a proper definition of *fact* is:

$$label[fact;\ \lambda[[x][eq[x;0] \rightarrow 1;\ t \rightarrow *[x;fact[sub1[x]]]]]]$$

To include *label* in the LISP syntax add:

<function>::= *label*[<identifier>;<function>]

and the S-expr translation of the *label* construct should naturally be:

$$\mathfrak{R}[\![label[f;fn]\ ]\!] = (LABEL\ \mathfrak{R}[\![f\ ]\!]\ \mathfrak{R}[\![fn\ ]\!])$$

Note that *label* is a special form, not a call-by-value function.

A typical application of the *label* construct, say $label[f;\lambda[[x]\xi[x]]][A]$. results in the following environmental picture when we get ready to evalute $\xi[x]$:

```
label[f;λ[[x]ξ[x]]][A]              ξ[x]
         E0                            E1
         | /                           | E0
                                 => ...
___________________                ______
         |                         f  | λ[[x]ξ[x]]
                                   x  | A
```

Notice that the definition does not appear in the global table E_0; we use *label* to create temporary

function definitions. These definitions disappear when the environment in which the *label* was executed is no longer accessible to the computation. Thus within the evaluation of the body $\xi[x]$ a recursive call on f will refer to the definition of f located in E_1 so long as f is not rebound in ξ; once we have completed the computation initialized in E_0 the definition of f will disappear.

With the introduction of *label* we can talk more precisely about "<=". When we write "what is the value of $f[2;3]$ when f <= $\lambda[[x;y]$ ...]?" we mean "evaluate the form *label*[f; $\lambda[[x;y]$ ...]][2;3]". If f is not recursive, then the use of *label* is unnecessary. The anonymous function $\lambda[[x;y]$ ...] applied to [2;3] suffices.

What about statements like "evaluate $g[A;B]$ where g <= $\lambda[[x;y]$ ... $f[u;v]$ ...] and f <= $\lambda[[x;y]$ ...]."? *label* defines only one function; we may not say *label*[f,g; ...]. What we can do embed the *label*-definition for f within the *label*-definition for g [62]. Thus:

$$label[g;\ \lambda[[x;y]\ \ldots\ label[f;\ \lambda[[x;y]\ \ldots\ ]][u;v]\ \ldots]]$$

Implementations of LISP offer better definitional facilities, with "<=" having the effect of permanently establishing the definition in E_0.

The apparent simplicity of the *label* operator is partly due to misconception and partly due to the restrictions placed on the current subset of LISP. *label* appears to be a rather weak form of an assignment statement. When we extend LISP to include assignments we can easily show that such interpretation of *label* is insufficient; when we talk about a mathematical interpretation of LISP we will show that *label* really requires careful analysis.

Problems

I. Show one way to change *eval* to handle *label*.

II. Express the definition of *reverse* given on page 43 using *label*.

III Evaluate the following:

$$\lambda[[y]label[fn;fn_2][f]]\ [f]$$

where:

$$fn_2 \Leftarrow \lambda[[x][y \to 1;\ x \to 2;\ t \to fn_1[t]]$$

$$fn_1 \Leftarrow \lambda[[y]fn[y]]$$

[62] Indeed every occurrence of f must be replaced by the *label*[f;...] construct.

3.10 Functional Arguments and Functional Values

Recall our discussion of :

$$eval[(F\ 2\ 3);((F\ .\ (LAMBDA\ (X\ Y)\ (PLUS\ (EXPT\ X\ 2)\ Y))))].$$

We now know this is equivalent to:

$$eval[((LABEL\ F\ (LAMBDA\ (X\ Y)\ (PLUS\ (EXPT\ X\ 2)\ Y)))\ 2\ 3);(\ )].$$

In either case, the effect is to bind the name f to the λ-expression. Binding also occurs when f is called: we bind x to 2, and y to 3. In the latter case we are binding simple values; in the former we are binding functions as values. We have decided that the necessary ingredients to characterize a functional value [63] are a representation of the formal parameters, and a representation of the expression described in the body of the function. In this section we will examine the adequacy of that decision. We will proceed informally with a few examples and see what happens.

Assume we have a list l of dotted-pairs $\alpha_1, ..., \alpha_n$, and we wish to form a new list of the form $(car[\alpha_1] ... car[\alpha_n])$. That is we wish to apply *car* to each of the elements of l. Such a function is easy to write:

$$carfirst \Leftarrow \lambda[[l][null[l] \rightarrow (\ );\ t \rightarrow concat[car[first[l]];carfirst[rest[l]]]]].$$

Now suppose we wish to write a more general function, which instead of being specific to *car*, will take an arbitrary unary function f and apply it to each of the elements of l, generating $(f[\alpha_1], ..., f[\alpha_n])$. Such a function could plausibly be defined as follows:

$$mapfirst \Leftarrow \lambda[[fn;l][null[l] \rightarrow (\ );\ t \rightarrow concat[fn[first[l]];mapfirst[fn;rest[l]]]]].$$

Thus the first calculation we requested above could be expressed as:

mapfirst[car;l] ....or could it?

Recalling LISP's penchant for call-by-value evaluation, we should realize that the computation would not be done as expected. We do not want the argument *car* evaluated. We want its **name**. We have seen one artifact in LISP to subdue evaluation: we can make it a constant by *quote*-ing it. Indeed,

mapfirst[quote[car];l] = *mapfirst[CAR;l]* will work.

You should convince yourself that *mapfirst[CAR;l]* will compute *carfirst[l]*; that exercise requires examining the details of *eval*.

---

[63] It would be better to call these constructs "procedure values" since we will take a decidedly algorithmic interpretation of them. As we now know, there are important differences between functions and algorithms.

Before going on to more complex examples it would be well to note that *mapfirst* is a different kind of LISP function from those we have seen before. The first argument to *mapfirst* is expected to be a function. Notice that the argument *fn* appears in the body of *mapfirst* in a position reserved for functions. Thus any actual parameter bound to *fn* is expected to be a representation of a function. Such a use of a function is called a **functional argument**.

The trick we used above of representing the functional argument *car* as a constant *CAR* can be applied to other instances of functional arguments.

Thus the functional argument:

λ[[x]f[g[x]] could be represented as,

(LAMBDA(X)(F(G X))).

The difficulty is that the trick [64] is not sufficient to capture the intended meaning in all cases. To understand why *QUOTE*-ing is not sufficient we need a slightly more complex set of examples. First we try:

mapfirst[quote[λ[[x]concat[;()]]];(A B C D)] [65] which we expect to evaluate to *((A)(B)(C)(D))*.

```
mapfirst[quote[λ[[x]concat[x;()]]]; ..]          [null[l] ...]
            E0                                       E1
            | /                                      | E0
     _______________                  =>         ______            => . . .
  mapfirst  | λ[[fn;l]][null[l]...]]              l  | (A B C D)
                                                  fn | quote[λ[[x]concat[x;()]]]
```

Now since *null[l]* is false, we [66] reduce the problem to:

```
        concat[fn[first[l]];mapfirst[fn;rest[l]]].
                  E1
                  | E0
        _______________
          l       | (A B C D)
          fn      | quote[λ[[x]concat[x;()]]]
```

---

[64] The trick is called *QUOTE*-ing the functional argument since the S-expr representation of an instance of such a construct is a *QUOTE*-ed expression.

[65] Note that we are continuing to use *quote* rather than write out the representation.

[66] When we say "we" we really mean *eval*.

Since we're doing call-by-value we have to evaluate the arguments to *concat*; that requires evaluating *fn[first[l]]*. The value of *l* we find locally and evaluate *first[l]*, getting *A*. The value for *fn* is also found locally, and since it is the representation of a λ-definition, we set up a new environment in which to evaluate the body of *fn*, binding the λ-variable *x* to *A*:

```
concat[x;()]
        E2
        | E1
____________
   x    | A
```

The expected evaluation takes place: *(A)* is computed and returned to environment E_1 so that we may continue the evaluation *mapfirst[fn;rest[l]]*.

However, consider the following variant of this last example. Define:

foo <= λ[[*l*]*mapfirst*[*quote*[λ[[*x*]*concat*[*x*;*l*]]]; *(A B C D)*].

It would seem that *foo[( )]* would also give *((A)(B)(C)(D))* since *l* will be bound to *( )* and therefore the *l* in the functional argument will effectively be *( )*.

```
foo[( )]          mapfirst[quote[λ[[x]concat[x;l]]]; ..]        [null[l] ... ]
     E0                          E1                                  E2
     | /                         | E0                                | E1
_________________      =>      _______              =>            _______      => ...
foo      | λ[[l]...             l  | ( )                          l  | (A B C D)
mapfirst | λ[[fn;l][null[l]...]]                                  fn | quote[λ[[x]concat[x;l]]]
```

null[l] is false since *l* is *(A B C D)*, so we evaluate *concat[fn[first[l]] ...]*. This involves evaluating *first[l]* in E_2, giving *A*. We evaluate (look up) *fn* in E_2 and, finding a representation of a λ-definition, we make a new environment E_3 in which to evaluate the body of *fn*. As we make E_3, we add an entry binding *x* to *A* and we settle down in E_3 to evaluate *concat[x;l]*:

```
  concat[x;l]
  E3
  | E2
___________
 x  | A
```

Since *l* is non-local to E_3, we follow the access chain to find its value in E_2 to be *(A B C D)*. But that's not the expected value! We expected to find *( )*, which was hidden away in E_1.

The trouble here is that *l* was rebound in the interim. The first thing to note is that the problem is

caused by free variables: *l* is free in the functional argument. Local variables aren't problematic; neither are global variables. Next note that the desired binding for *l* is the one which was current when we were binding the functional argument to the formal parameter *fn*. A plausible solution then is to replace all non-local variables with their values at the time we recognize the functional argument. This will not suffice. A sufficient solution is to associate the name of the current environment with the function and use that pair as the value to be given to the formal parameter. When we want to apply the functional argument we set up a new environment as always, introducing a local table with the λ-variables bound to their values; only now we use the saved environment as the beginning of the access chain. Then the values of any non-local variables which we encounter in the process of applying the functional argument will be searched for in the saved environment.

To initialize this process we require the recognition of instances of functional arguments. We introduce a new operator called *function*. This operator takes one argument: a representation of the function. The effect of *function* will be to construct a value representing that argument and the environment which was current when the *function*-instance was evaluated.

Thus in the example we would recognize the *function*-construct while evaluating the arguments to *mapfirst*; the environment which was current then was E_1. Therefore as we build E_2 we want to associate the pair λ[[*x*]*concat*[*x*;*l*]] - E_1 with the formal parameter *fn*. Whenever we apply *fn* we want to use λ[[*x*]*concat*[*x*;*l*]]; and within that context, whenever we want *l*, we want the value of *l* in E_1.

The function-environment pair is called a **closure** or **funarg**. In our diagrams we will designate the pair as:

<function>:<environment>.

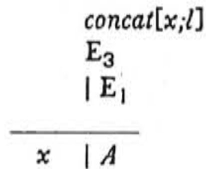
Thus in the above example we should designate the value of the functional argument as:

λ[[*x*]*concat*[*x*;*l*]]:E_1.

We must also extend the manipulation of Weizenbaum environments to handle such constructions. The process which recognizes λ-definitions and sets up new environments must now watch for funargs. When it sees one it uses the associated environment as the access environment. Let's do the example again.

```
     foo[( )]      mapfirst[function[λ[[x]concat[x;l]]; ..]      [null[l] ...  ]
          E0                              E1                              E2
          |/                              | E0                            | E1
________________          =>         ______             =>         ______      => . . .
     foo    | λ[[l]...                l   | ( )                    l   | (A  B  C D)
mapfirst    | λ[[fn;l][null[l]...]]                                fn  | λ[[x]concat[x;l]]:E1
```

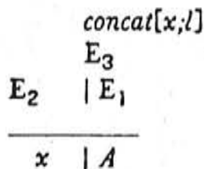
Things are as before except now *fn* is bound to the funarg pair in E_2. We look up *fn* in E_2 and, finding a λ-definition, we make a new environment E_3 in which to evaluate the body of *fn*. As we make E_3, we add an entry binding *x* to *A*. But now since the λ-definition is a funarg we make the access environment E_1 as saved with *fn*. Thus we settle down in E_3 to evaluate *concat*[x;l]:



Since *l* is non-local to E_3, we follow the access chain to find its value in E_1 to be *( )* as desired. Thus instead of simply tracing back to the previous environment we detour around E_2:

However, there is still some information which we must make explicit if these Weizenbaum diagrams are to faithfully represent the process of evaluation. Namely, after we have finished the evaluation of *concat*[x;l] we are to restore a previous environment. Which one is it? It isn't E_1, it's E_2! That information is not available in our diagram, so we must correct the situation.

In the left-hand quadrant of our diagram we place the name of the environment which we wish restored when we leave the current environment. That environment name will be called the **control environment**, and will head a chain of environments, called the control chain [67]. Here's the correct picture:



[67] In Algol, the access chain is called the static chain, and the control chain is called the dynamic chain.

So after we have finished the computation in E_3 we return control to E_2. Thus the general structure of an environment is as follows:

$E_{control}$	Form $E_{current}$ E_{access}
var	value
x_1	...
x_2	...
...	...
x_n	...

Here's another example, involving a function to produce the composition of two unary functions. We will call the function *compose*. The value returned by *compose* will be a function. Thus *compose* will produce functional values:

compose[function[car];function[cdr]] = *cadr*

with a plausible definition as:

compose <= λ[[*f*;*g*]λ[[*x*]*f*[*g*[*x*]]]]

This definition of *compose* is almost right. The value returned by *compose* is to be a function. Indeed it is an instance of a functional value, so, as with functional arguments, it needs to be decorated with *function* so that the environment which contains the right bindings for *f* and *g* is saved. Which environment is that? It's the environment which will be current when the *function*-construct is recognized. So we write:

compose <= λ[[*f*;*g*]*function*[λ[[*x*]*f*[*g*[*x*]]]]].

Now try: *app[cons[A;(B . C)];compose[function[car];function[cdr]]]*
where: *app* <= λ[[*y*;*f*]*f*[*y*]]

As usual we evaluate the arguments to *app*, bind the results to *y* and *f* and evaluate the body of *app*.

app[cons[A;(B .C)];function[compose[function[car];function[cdr]]]]

E_0 / | /	
app	λ[[*y*;*f*]*f*[*y*]]
compose	λ[[*f*;*g*]*function*[λ[[*x*]*f*[*g*[*x*]]]]]

Evaluation of the first argument to *app* brings no surprises; we get *(A .(B . C))*. We begin evaluating the second argument; we find the definition of *compose* in the environment and since it is a λ-definition we set up a new environment, E_1, and evaluate the body *function*[λ[[x]f[g[x]]]]:

function[λ[[x]f[g[x]]]]

$E_\emptyset$	E_1 $E_\emptyset$
f	*car*:$E_\emptyset$
g	*cdr*:$E_\emptyset$

Again, the recognition of the *function*-construct says return a funarg-pair as value. The environment we associate is the current one, E_1. We now go back to $E_\emptyset$, using the control chain. Since both arguments to *app* are now evaluated, we find the definition of *app* and set up a new environment E_2. Thus:

f[y]

$E_\emptyset$	E_2 $E_\emptyset$
y	*(A. (B . C))*
f	λ[[x]f[g[x]]]:E_1

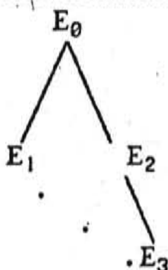
=>

f[g[x]]

E_2	E_3 E_1
x	*(A. (B . C))*

The form to be evaluated in E_2 is f[y]; we find y and f both locally. We evaluate the argument y, then since f is a λ-definition, we set up a new environment binding the λ-variable x to the value *(A .(B . C))*. But the λ-definition is also a funarg; therefore the access environment stored in E_3 is E_1. The control component of E_3 is set to the prior environment, E_2; and we begin evaluation of the body f[g[x]].

Now in E_3 we find x locally but have to resort to the access chain to find f and g; using funargs, we have set up the appropriate environments. From E_3 we have access to E_1 [68]:



The rest of the evaluation goes without a hitch: we finish the evaluation in E_3 and return to E_2 and finally to $E_\emptyset$ following the control evironments.

[68] and from there to $E_\emptyset$ if it were needed.

Notice that f and g in the body of *compose* are free variables and therefore their bindings are not to be found in the local environment. Since the interesting applications of such functions usually involve free variables, we must deal with them. In particular, the *label* operator will typically involve free variables. We remarked that f in:

$$f \Leftarrow \lambda[[x][zerop[x] \rightarrow 1;\ t \rightarrow *[x;f[x-1]]]\ ]$$

is free. But we want the occurrence of f on the right to be synonymous with the f being defined on the left. We can do this by "tying a knot" in the access environment chain. Thus we should modify the diagram for *label* to be:

```
label[f;λ[[x]ξ[x]]][A]              ξ[x]
         E0                              E1
         |/                              | E0
                            => ...
______________                      ______
         |                          f   | λ[[x]ξ[x]]:E1
                                    x   |A
```

Since every language construct in LISP must have an S-expr representation we must include the *function*-construct. Its translation scheme is simple: represent *function*[ξ] as *(FUNCTION* $\Re[\![\xi]\!]$*)*.

Thus:

function[λ[[x]*f*[*g*[*x*]]] has an $\Re$-image of,

(FUNCTION(LAMBDA(X)(F(G X)))).

Similarly we must develop parts of *eval* to deal with *FUNCTION*. The device LISP used to associate environments with functions is called the *FUNARG* device. (More is said about implementations of *FUNARG* in Section .) When *eval* sees the construction *(FUNCTION* fn*)* it returns as value the list:

(FUNARG fn current-symbol-table*)*.

When *eval*, or actually *apply* sees *(FUNARG* fn st*)*, that is, when we are calling *fn*, we use the symbol table *st*, rather than the current symbol table for accessing free variables.

Thus there are two environments involved in the proper handling of functional arguments. First there is the environment which is saved with the *FUNARG*. This is called the **binding environment** since it is the environment current at the time the functional argument was constructed or bound. The second environment, called the **activation environment**, is the environment which is current when the functional argument is applied or activated. Thus the activation environment is used to locate local variables, but if a non-local variable is needed then the binding environment is selected.

It is the duty of *eval* and *apply* to use the *FUNARG* device to maintain the proper control of the activation and binding environments.

Finally, we should update our description of the usage of Weizenbaum environments given on page 110:

> When the *function* construct is recognized, we manufacture a *FUNARG* triple consisting of the atom *FUNARG*, the function described in the instance of *function*, and the current environment. This triple is the value of the *function* construct and may thus be bound to any LISP variable; typically the LISP variable will appear in an expression in a position reserved for functions.
>
> When doing a λ-binding, set up a new E_{new} with the λ-variables as the local variable entries and the values of the arguments as the corresponding value entries. The control slot of the new E_{new} always points to the previous symbol table. The access slot also points to the previous environment unless the function being applied is a *FUNARG*. If it is a *FUNARG*, then set the access slot to the environment which was saved with the *FUNARG*.
>
> The evaluation of the body of the λ- expression takes place using E_{new}; when a local variable is accessed we find it in E_{new}; when a non-local variable occurs, we chase the access chain to find its value. When the evaluation of the body is completed, the previous environment is restored. E_{new} disappears unless the value of the computation is a functional value.

Notice that there is a certain asymmetry about access and control. The control slot always points at the previous environment, while the access slot may vary. It may follow control, as is the case on simple function calls; it may point to an environment further down the control chain, as is the case for functional arguments; it may point to an environment which control cannot return to, as is the case for functional values; or it may point to itself as is the case for *label*'s implementation.

There is another asymmetry in the properties of access and control. The access environment is a self-sufficient data structure; it can be described and manipulated as such using the usual constructors, selectors, and recognizers. Typically such environments come into existence as a part of a computation; they are constructed during the λ-binding process. We can implicitly save such an environment through the *FUNARG* device; and we can explicitly build such environments using the data structure operations and pass them to *eval* as a symbol table. But symbol tables are independent of the method used to create them. In particular, once a table has been captured by a *FUNARG* we need not retain any information about the computation which created that table. However the idea of "control" and "state of computation" is integrally tied to access structure. The state of the computation involves the expression currently being evaluated, the history of those computations which are suspended and waiting for the completion of the current computation, and it also involves the access evironment since that is necessary for the crrect evaluation of variables. To "save the state of computation" implies saving the partial computation to that point, saving the expression being evaluated, and saving the current access environment.

Thus to a large extent "control environment" is a misnomer. What we are intending to capture is the idea of a suspended computation: suspended until the subsidiary computation has been completed. Part of the suspended computation is the "control environment", but there's more. The Weizenbaum diagrams show part of the information; they show the environments and the

expressions being evaluated. However they leave implicit the dynamics of the computation: which argument is being evaluated, and where are the partial results being stored, and where in the expression we are to continue when the subsidiary computation is completed. In Section we will develop a different *eval* family which will make much of this information explicit. Also in Section we will examine the possibility of expanding the behavior of control slots. That is, allowing environments other than the predecessor to appear in the control slot of an environment.

What does all this say about functions? We have already remarked that functions are parametric values; to that we must add that functions are also tied to the environment in which they were created; they cannot be evaluated *in vacuo*. What does this say about "<="? It still appears to act like an assignment statement. It is taking on more distinct character since it must associate environments with the function body as it does the assignment.

The correct implementation of the semantics of *function* seems like a lot of work to allow a moderately obscure construct to appear in a language. However constructs like functional arguments appear in several programming languages under different guises. Usually the syntax of the language is sufficiently obfuscatory that the true behavior and implications of devices like functional arguments is misunderstood. Faulty implementations usually result. In LISP the problem **and the solution** appear with exceptional clarity [69].

Here is a sketch of the abstract structure of the current *eval*.

```
eval <= λ[[exp;environ]
            [isvar[exp] → value[exp;environ];
             isconst[exp] → denote[exp];
             iscond[exp] → evcond[exp;environ];
             isfun[exp] → makefunarg[exp;environ];
             isfunc+args[exp] → apply[func[exp];evlis[arglist[exp];environ];environ]]
             ]
```

where:

```
apply <= λ[[fn;args,environ]
            [isfunname[fn] → ...;
             islambda[fn] → eval[body[fn];newenviron[vars[fn];args;environ]];
             isfunarg[fn] → apply[func_1[fn];args;evn[fn]];

                   ...          ... ]]
```

The reader is encouraged to complete the definitions, supplying appropriate constructors, selectors and recognizers.

Now for some specific examples. In particular, most implementations of LISP include a very useful class of mapping functions.

---

[69] Indeed, LISP was the first language to allow functional values.

maplist is a function of two arguments, a list *l* and a functional argument *fn*. *maplist* applies the function *fn* (of one argument) to the list *l* and its tails (*rest*[*l*], *rest*[*rest*[*l*]], ..) until *l* is reduced to ( ). The value of *maplist* is the list of the values returned by *fn*. Here's a definition of *maplist*:

maplist <= λ[[*fn*;*l*][*null*[*l*] → ( ); t → *concat*[*fn*[*l*];*maplist*[*fn*;*rest*[*l*]]]]].

Thus:

maplist[*function*[*reverse*];(*A B C D*)] = ((*D C B A*)(*D C B*)(*D C*)(*D*)). [70]

An interesting and non-trivial use of functional arguments is shown on page where we define a new control structure suitable for describing algorithms built to operate on lists.

Problems

I. What changes should be made to the LISP syntax equations to allow functional arguments?

II. Use *app* on page 118 to define a function which computes factorial without using *label* or explicit calls on the evaluator.

III. Extend *eval* and friends to handle functional arguments.

IV. An interesting use of functional arguments involves self-applicative functions. An application of a function *f* in a context *f*[...;*f*;...] is an instance of self application [71]. Self-applicative functions can be used to define recursive functions in such a way that the definition is not statically self-referential, but is dynamically re-entrant. For example, here is our canonical example, written using a self-applicative function:

fact <= λ[[*n*]*f*[*function*[*f*]; *n*]]

f <= λ[[*g*;*n*][*n*=*0* → *1*; t → *[*n*; *g*[*g*; *n*-*1*]]]]

Use Weizenbaum's environments to show the execution of *fact*[2].

[70] *quote*-ing *reverse* would also work since *reverse* uses no free variables.

[71] provided the designated argument position is a functional argument.

3.11 Binding strategies

After the discussion of variables in Section 3.7 and the intervening discussions of environments, it should now be clear that the root of the binding problem is free variables. We would rather not outlaw free variables; many interesting recursive algorithm have free variables. We don't want to restrict the use of free variables too precipitously since they are a very useful programming technique. For example, the possible alternative of passing all global information through as extra parameters in calling sequences is overly expensive [72].

Handling of free variables varies from programming language to programming language. The solution advocated by Algol-like languages is called static binding and dictates that all non-local references be fixed in the binding environment; thus free variables aren't really free in the sense that we have a choice to make. LISP at least gives you a choice. Using *quote* you will get the dynamic binding on free variables in a functional argument; using *function* gives the static interpretation [73]. There are no questions about Algol's interpretation of functional values: the construct is not allowed. When we discuss implementation we will see why.

3.12 special forms

We have remarked that the evaluation scheme for LISP functions is call-by-value and, for functions with multiple arguments, left-to-right evaluation of arguments. We have also seen, in *quote* and *cond*, that not all forms to be evaluated in LISP fall within this category. We have already noted on page 86 that *QUOTE* and *COND* are not translations of functions in the LISP sense. Indeed the purpose of *quote* was to stop evaluation. Also, the "argument list" to *cond* is handled differently; its evaluation was handled by *evcond*. Since *quote* and *cond* were rather anomalous we have called them special forms. However, now we would like to discuss special forms as a generally useful technique.

Consider the predicates *and* and *or*. We might wish to define *and* to be a binary predicate such that *and* is true just in case **both** arguments evaluate to t, and define *or* to be binary and false just in case both arguments evaluate to f. Notice two points. First, there is really no reason to restrict these predicates to be **binary**. Replacing the words "binary" by "n-ary" and "both" by "all" in the above description has the desired effect. Second, if we evaluate the arguments to these predicates

---

[72] Though much of that expense can be mitigated by a clever compiler.

[73] A case can be made for even more flexibility in the interpretation of free variables. We could ask that the binding be done on a per variable basis. That is we could declare which free variables are to be captured statically and which are to be captured dynamically. We could also ask that both bindings be available and supply selectors which would access either the dynamic or the static binding.

in some order, say left-to-right, then we could immediately determine that *and* is false as soon as we come across an argument which evaluates to f; similarly a call on *or* for an arbitrary number of arguments can be terminated as soon as we evaluate an argument giving value t. But if we insist that *and* and *or* be LISP functions we can take advantage of neither of these observations. Rather we will define *and* and *or* as special forms and handle the evaluation ourselves. Presently, the only way to handle special forms is to make explicit modifications to *eval*. In Section , we will introduce a simple way to add such forms without modifying *eval*. See also page .
Recognizers for the predicates must be added to *eval*:

isand[*e*] → *evand*[*args*[*e*];*environ*];
isor[*e*] → *evor*[*args*[*e*];*environ*];
where:

```
evand <= λ[[l;a]   [null[l]→ t;
                    eval[first[l];a] → evand[rest[l];a];
                    t → f] ]

evor <= λ[[l;a]    [null[l] → f;
                    eval[first[l];a] → t;
                    t → evor[rest[l];a]] ]
```

Notice the explicit calls on *eval* [74]. This is expensive, but cannot be helped. Later we will show a less costly way to handle those "non-functions" which have an indefinite number of arguments, all of which are to be evaluated (see Section on macros).

Problems

I What is the difference between a special form and call-by-name? Can call-by-name be done in LISP (without redefining *eval*)?

II *select* is a special form to be called as: *select*[$q;q_1;e_1; \ldots ;q_n;e_n;e$] and to be evaluated as follows: q is evaluated; the q_i's are evaluated from left to right until one is found with the value of q. The value of *select* is the value of the corresponding e_i. If no such q_i is found the value of *select* is the value of e. *select* is a precursor of the case statement; see page 133. Add a recognizer to *eval* to handle *select* and write a function to perform the evaluation of *select*.

[74] Also notice that the abstract versions of *evand* and *evor* know that the arguments are also presented as a sequence. The structure of the recursion implies a left-to-right evaluation.

3.13 The *prog*-feature

Though recursion is a significant tool for constructing LISP programs, there is another technique for defining algorithms in LISP. It is an iterative style of programming which is called the *prog* or *prog*ram feature.

Many algorithms present themselves more naturally as iterative schemes. Recall the recursive algorithms *length* and $length_1$ given on page 43. These algorithms computed the length of a list. Compare those schemes with the following:

1. Set a variable *l* to the given list. Set a variable *c* to zero.

2. If the list is empty, return as value of the computation, the current value in *c*.

3. Otherwise, increment *c* by one.

4. Set *l* to the *rest* of *l*.

5. Go to line 2.

Here is a LISP *prog* version of the algorithm:

```
length <= λ[[x]prog[[l;c]
                  l ← x;
                  c ← 0;
                a [null[l] → return[c]];
                  c ← c+1;
                  l ← rest[l];
                  go[a]] ]
```

We have introduced several new symbols, formats, and functions in this example. These innovations must be explained before the example is complete. First, the basic syntax of a *prog* is given by:

```
<prog>          ::= prog[[<prog variables>]<prog body>]
<prog body>     ::= <prog element><prog body> | <prog element>
<prog element>  ::= <label> | <prog form>;
<label>         ::= <identifier>
<prog form>     ::= <application>
                ::= <conditional statement>
                ::= <assignment statement>
                ::= <return statement>
                ::= <go statement>
```

<conditional statement>	::= <conditional expression>
<assignment statement>	::= <identifier> ← <form>
<return statement>	::= *return*[<form>]
<go statement>	::= *go*[<form>]

The *prog* variables, *l* and *c*, in the example, are local variables. They act just like λ-variables with an implied initialization to *( )*. Thus *progs* can be used recursively.

The *prog* body is a sequence of *prog* forms and labels. Each *prog* form is evaluated in the usual LISP manner, and since the *prog* body can consist of a sequence of *prog* forms, the *prog*-body is evaluated from left-to-right.

If the intent of the *prog* was simply to execute the sequence of *prog* forms, in left-to-right order, then *prog* could be replaced by a much simpler construct like *progn*:

$$progn \Leftarrow \lambda[[x_1; ...; x_n]\ x_n]$$

However we will add constructs to LISP which will allow us to vary the flow of control within the *prog* body. It is to this end that we use labels. Before we discuss labels and their associated operations, we wish to discuss the more general character of *prog* forms. In LISP, every form returns a value; programming languages also have constructs which are primarily executed for their effect rather than their value. Such constructs are called **statements**. For example, LISP implementations include a unary primitive named *print* whose effect is to print the value of its argument on the current output device. This function also returns its evaluated argument as the value of the *print* statement. Thus mathematically, *print* acts like an identity function, but its execution certainly affects the programmer's environment. Operations which can have such non-local effects are said to have **side-effects**[1].

There is another common and useful programming construct we wish to introduce into *progs* which has side-effects. It is called the **assignment statement**. As with all LISP constructs, the assignment returns a value, but we identify it as a statement since it is executed more for effect than for value.

In our example of *length*, we used an assignment to bind *l* to the value of *x* and to bind *c* to *0*. To evaluate an assignment, we first evaluate the form; then the identifier is located by searching the access chain. Thus the identifier may be a non-local variable. When the identifier is located its current value is replaced by the value of the form. Notice that this is a different kind of binding than that previously done by λ-binding. In λ-binding we always associated a new value with a newly created local symbol table as we entered the λ-body. We never changed the binding of a variable, though we could achieve the effect by rebinding the variable. The semantics of the assigment involve changing the binding. Thus assignments to non-local variables can have effect outside the *prog*. Assignment statements therefore have a side-effect. An assignment statement also has a value. Its value is the value of the form on the right-hand-side.

---

[1] Whether the act of printing is a side-effect or the fact that *print* returns a value is a side-effect depends on your point of view.

As we intimated earlier, *prog* introduces some new control structures so that the *prog* body need not be executed in simple left-to-right order. The control structures are: the conditional statement, the *return* statement, and the *go* statement.

Though conditional statements in *progs* have the same syntax as conditional expressions, their semantics is slightly different. A conditional statement is executed in the usual manner unless none of the predicate alternatives is satisfied. Recall that a conditional expression is undefined in this case; a conditional statement however is defined, returns ( ), and executes the next statement in the *prog* body. Thus in our *length* example:

$$[null[l] \rightarrow return[c]],$$

if l is not empty the *prog* body continues at the next statement with the assignment of *rest*[l] to l. If l is empty, then the statement *return*[c] is executed.

The *return* statement is a *prog* construct similar in effect to exiting a λ-expression. It is used to leave a *prog* body and return to the caller of the *prog*. As we leave the *prog*, the bindings of the *prog* variables are removed as are the λ-bindings made on entry to the *prog*. The value returned is the value of the argument to the *return* statement. The *return* statement may be nested within other LISP computation, as for example:

$$concat[A;return[list[B]]].$$

However the evaluation of the *return* is to take effect immediately; the *concat* would never complete its operation. We would return *(B)* to the caller of the enclosing *prog*. A bit of care is needed in describing the meaning of *return*: we look for the latest instance of an entrance to a *prog* and return from that *prog*. The simplest way to visualize this is to use Weizenbaum environments (page 117. We search the **control chain**, looking for the first Form which is *prog*[[...] ...]. We then restore using the access and control information found in that diagram. We will give a comprehensive example after discussing the *go* statement.

The *go* statement is used in conjunction with labels to divert the implied left-to-right execution of the *prog* body. Labels really aren't executed; they are used to name statements in a *prog*. It is the *go* statement which uses the label as a destination for transferring control. Labels may conflict with the λ-variables or *prog* variables since the evaluator for *progs* can resolve the conflicts by context. Any identifier occurring by itself in a *prog* body is a label. Any identifier occurring in an application other than a *go* statement is a variable and its value is searched for in the access chain, whereas an identifier appearing in a *go* statement is interpreted as a label and searched for in a *prog* body.

The *go* statement is a little more complicated than the *return* statement. If the argument to *go* is an identifier then it is interpreted as a label; otherwise, the argument is evaluated and the result of the evaluation is interpreted as a label. Once a label has been uncovered as the result of this evaluation we must locate a statment in a *prog* which has that label attached to it. Our intention is to transfer control to that statement. We locate the labeled statement as follows: we look back through the control chain for the first *prog* which contains the label. When the label is found we transfer control to that labeled statement, restoring the access and control environments of the *prog*

which contain that statement. Thus there is a double search involved: we search the control chain for *prog* forms, and search the *prog* forms for the label. Labels need not be local; we find the closest dynamically surrounding *prog* which contains the label.

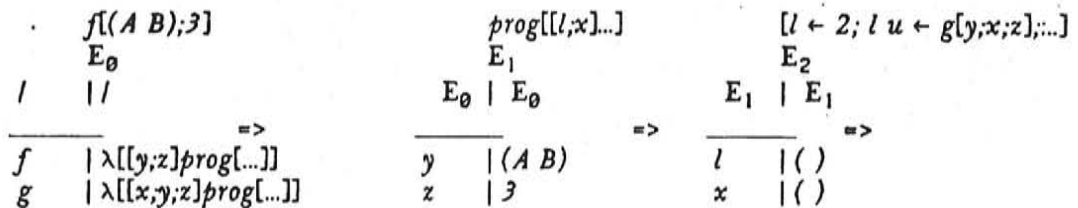
Notice that *go* and *return* are the first constructs we have seen whose behavior on completion does not imply that we restore the previous control environment.

Finally, as an example covering the new features of *prog* consider:

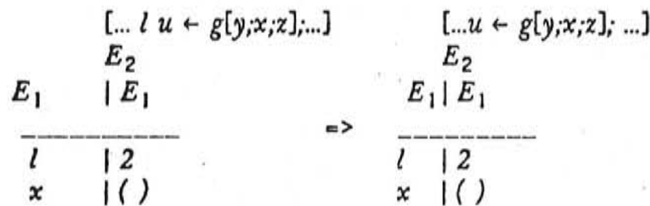
```
f <= λ[[y;z]prog[[l;x]
                 l ← 2;
               l u ← g[y;x;z]
                     ...
                 ]]

g <= λ[[x;y;z]prog[[ ]
                   ...
                   go[l]
                   ...
                   return[first[x]]
                   ...
                 ]]
```

Notice in *f*, *l* is both a label and a *prog* variable. Notice in *g* that we have no *prog* variables; and since we assume that *l* is not a label in *g* we have a non-local *go*.

Consider the evaluation of *f*[(*A B*);*3*].



At this point we have done the λ-binding and initialized the *prog* variables. As we begin the execution of the *prog* body, we assign *2* to *l* and, since labels have no computational effect, begin the evaluation of the assignment statement: *u* ← *g*[*y*;*x*;*z*]:



We evaluate $g[y;x;z]$:

```
              prog[[ ] ...]           [...go[l]; ... return[first[x]]; ...]
              E3                          E4
     E2     | E2                 =>    E3| E3
     ------------------                ---------
     x      | (A B)
     y      | ( )
     z      | 3
```

The $go[l]$ will search the control chain; it looks at the *prog* of E_3 but finds no label l. It finds the *prog* of E_1 next, and there it does find the label l. Thus execution would be continued in E_2, the environment which bound the *prog* variables, at the assignment statement. In general, we continue in the envirnment which was created on entry to the *prog* body.

Notice that once we have left E_4 there is no way to jump back into it. We can only search down the control chain, and the entry to g is not below that of f on that chain. An extension of the semantics of LISP could allow such generalized control and we will develop some of those ideas in Section .

If we executed the $return[first[x]]$ in E_4 an action similar to that of *go* would transpire. We would evaluate $first[x]$, getting A. We would search the control chain for the latest *prog* expression; here found in E_3; and then return control to the environment designated in the control quadrant; here E_2. Thus we return A as the value of $g[y;x;z]$. Since the call on g was a component of the assignment $u \leftarrow g[y;x;z]$, we must complete that assignment. We search the access chain for u. Since u is not found we make a global assignment in E_0:

```
          E0
          /|/
   ----------------
    f   | λ[[y;z] ...]
    g   | λ[[x;y;z] ...]
    u   | A
```

The ability to evaluate the argument to *go* results in a useful programming trick. Let l be a list of dotted pairs, each of the form, ($object_i$. $label_i$). At each $label_i$ we begin a piece of program to be executed when $object_i$ has been recognized. Then the construct:

UGH $\qquad go[cdr[assoc[x;l]]]$

can be used to "dispatch" to the appropriate code when x is one of the $object_i$. This is an instance of **table-driven** programming. The blocks of code dispatched to can be distributed throughout the body of the *prog*. Each block of code will usually be followed by a *go* back to the code involving equation UGH (above). In fact the argument l in UGH may be global to the *prog*-body. The effect

is to make a *prog* which is very difficult to understand. The LISP *select* (page 125) will handle many of the possible applications of this coding trick and result in a more readable program. The case-statement (page 133) present in some other languages is also a better means of handling this problem.

The *go* statement is useful if used with discretion. It is a building block for constructing more complex control regimes, particularly since the label need not be local to the *prog*. We will examine some more complex kinds of control behavior in Section .

Now to the problem of translating a *prog* into an S-expression representation: the construct,

$prog[[v_1; ...; v_n] ...\ ]$ will be translated to:

(PROG(V1 ... VN) ...).

Notice that *prog* is a special form. So the body of the *prog* must be handled specially by a new piece of the evaluator.

Similarly we must be careful about the interpretation of ←. We will write $x \leftarrow y$ in prefix form: *setq*[*x;y*]. We will map this to:

(SETQ X Y).

Notice that *setq* is also a special form. For if *x* and *y* have values *2* and *3*, for example, then the call-by-value interpretation of *setq*[*x;y*] would say *setq*[*2;3*]. This was not our intention. We want to evaluate the second argument to *setq* while stopping the evaluation of the first argument.

LISP has another assignment-like operator called *set*. Both arguments of this binary operator are evaluated; the value of the first argument is expected to be a representation of a variable; that is, the first argument evaluates to a literal atom. The second argument is a LISP form and using the value of that form, an assignment is made to the variable represented by the first argument. Thus we could define *setq* as:

setq[*x;y*] = *set*[*quote*[*x*];*y*]

As a more complex example, consider *set*[*z; plus*[*x;1*]]. If the current value of variable *z* is an identifier, then *set*[*z; plus*[*x;1*]] makes sense. Assume the current value of Z, the representation of *z*, is *A*; and assume the current value of *x* is *2*; then the effect of the *set* statement is to assign the value *3* to *a*.

Normally when you are making assignments, you want to assign to a **name** and not a **value**; thus you will tend to use the *setq* form.

Finally, here is a translation of the body of the *prog* version of *length:*

```
(LAMBDA (X)
        (PROG (L C)
           (SETQ L X)
           (SETQ C 0)
        A  (COND ((NULL L) (RETURN C)))
           (SETQ C (ADD1 C))
           (SETQ L (REST L))
           (GO A) ))
```

Now that assignment statements are out in the open let's re-examine "<=". We already know (page 122) that "<=" does more than simply associate the right hand side with a symbol table entry of the left hand side; it must also associate an environment with the function body, and this environment is to be used for accessing non-local variables. This operation of associating environments is called forming the **closure**. We thus might be tempted to say:

f <= λ[[...] ...] is $f \leftarrow$ *function*[λ[[...] ...]].

Alas, this implementation is still not sufficient as we will see in Section .

Problems involving *prog*

I. Write *prog*-versions of the following functions (or predicates).

1. *member*[x;y] <= ... : x is atomic; y is a list of atoms. *member* is to return t just in the case that x is one of the elements in y.

2. The factorial function.

3. *delete*[x;y] <= ... : x is atomic; y is a list of atoms. *delete* is to return a list which looks like y, except all occurrences of x have been deleted.

4. The *append* function.

5. *last*[x] <= ... : x is a non-empty list. *last* is to return the last element in x.

6. Now write the S-expr translations of each of your functions.

II. What is necessary to extend the evaluator to recognize *prog* and friends?

III. The *go*[*cdr*[...]]-construct on page 139 is better handled with a **case statement**. A typical syntax for such might be:

case<index>of <form$_1$>; ...<form$_n$>.

<index> is to evaluate to an integer, i. Where $0<i\leq n$. The ith <form> of the case-statement is executed, and is the value of the statement. Construct a reasonable case statement and extend the evaluator to recognize it.

IV. Some languages allow constructs like:

(**if** p(x) **then** x **else** y) ← exp, which is to mean the same as:

if p(x) **then** x← exp **else** y ← exp

Can such a construct be written in LISP?

V. Compare the *prog* version of *length* on page 126 with $length_1$ on page 43. Do you see any interesting relationships?

λ-notation 89
and 124
append 44
apply 95
atom 17
car 12
car-cdr-**chains** 13
cdr 12
concat 27
COND 86, 93
cons 11
eq 17
equal 21
eval 93
evcond 93
FUNARG 120
function 116
go 128
isseq 26
label 111
list *32*
maplist *123*
mkent *88*
NIL *30*
null *25*
or *124*
prog *126*
prog variables *127*
QUOTE *85*
rest *26*
reverse *44*
seq 27
tgmoaf 75
tgmoafr 75
a-lists 87
access chain 107
access link 107
activation environment 120
actual parameters 14
assignment 126
assignment statement 127
association lists 87
auxiliary function 43
bind 83, 95
binding environment 120
binding strategy 124
box-notation 8
call-by-name 82
call-by-value 82
case statement 133
case statement 125
closure 116, 132
conditional expression 16, 93
constructor 11
control environment 117
control structures 36
definition by recursion 39
differentiation 51
discriminator 17
dotted-pair 5
dynamic binding 106
evaluation 80
examples of *eval* 96
Fibonacci sequence 42
form 16, 90
formal parameters 13
forms 12
free variable 104
funarg 116
function 90
function application 13
functional argument 113
functional composition 12
functional value 118
global variable 104, 106
inductive definition 2
internal lambdas 92
lambda variables 90
left-hand values 133
length 126
linear search 88
list 30
list terminator 30
list-notation 30
lists 28
literal atoms 4
local variable 104, 127
mapping functions 122

match-variable 6
monotonic functions 20
non-local variable 104
non-strict 19
partial function 10
partial functions 12
polymorphic functions 26
predicates 16
prefix notation 52
recognizer 17, 26
S-expressions 4
S-exprs 4
selector 12
self-applicative functions 123
side-effects 127
special form 124
special forms 93
statement 127
static binding 124
strict functions 11
symbol tables 87
Symbolic expressions 4
table-driven 131
termination conditions 41
total function 10
type fault 21
unbound variable 104
variables 104