FORMAL PROOFS CONCERNING PARTIAL RECURSIVE FUNCTIONS

by

LEO JOSEPH ROTENBERG

S.B., Massachusetts Institute of Technology
1965

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June, 1966

Signature of Author

Department of Electrical Engineering, May 20, 1966

Certified by

Thesis Supervisor

Accepted by

Chairman, Departmental Committee on Graduate Students
FORMAL PROOFS CONCERNING PARTIAL RECURSIVE FUNCTIONS

by

LEO JOSEPH ROTENBERG

Submitted to the Department of Electrical Engineering
on May 20, 1966 in partial fulfillment of the requirements
for the degree of Master of Science

ABSTRACT

We have developed a formal system in which proofs concerning
 recursiveness of functions defined in LISP may be constructed. This
system includes a language, an interpretation, and a set of axioms
and rules of inference. We present a number of formal proofs
expressed in our formal system.

We have discovered a mechanical procedure which can determine
the domain of a function, for a large class of functions. Given
the definition of a function in LISP, the program produces the
LISP definition of a characteristic function of the domain of the
given function. If the given function is potentially k-recursive,
then this procedure produces a recursive function.

Thesis Supervisor: Marvin L. Minsky
Title: Professor of Electrical Engineering
A long moment of English silence.

The English clock strikes 17 English strokes.
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>INTRODUCTION</td>
<td>5</td>
</tr>
<tr>
<td>II</td>
<td>THE FORMAL SYSTEM</td>
<td>7</td>
</tr>
<tr>
<td>III</td>
<td>SOME FORMAL PROOFS</td>
<td>20</td>
</tr>
<tr>
<td>IV</td>
<td>POTENTIAL RECURSIVENESS</td>
<td>31</td>
</tr>
<tr>
<td>V</td>
<td>A MECHANICAL PROCEDURE FOR FINDING CHARACTERISTIC FUNCTIONS</td>
<td>42</td>
</tr>
<tr>
<td>Appendix A</td>
<td>THE DOMAIN PROGRAM</td>
<td>50</td>
</tr>
<tr>
<td>Appendix B</td>
<td>EXAMPLES</td>
<td>54</td>
</tr>
</tbody>
</table>

Bibliography 57
Chapter I

INTRODUCTION

In the following chapters we develop a formal (axiomatic) system in which one can express proofs concerning recursiveness of functions defined in LISP. The formal system includes a language (which contains the data- and meta-languages of LISP), a standard interpretation of that language (which relies on Church's thesis and an intuitive notion of "S-expression"), a (recursive) set of axioms, and the usual rules of inference. Since one of our predicates is effectively a quantifier over function symbols, this formal system is a second-order logic. The system was designed to allow fairly free-swinging representations of completely formal proofs.

Our major interest is proofs of statements of the form, "if p[x], then e converges", where p[x] is a predicate and e is a form. Such proofs are of particular interest in the context of a mathematical theory of computation, where we wish to make strong assertions about particular computations (or programs). Furthermore, formal proofs are necessary because there is no effective procedure for determining (for example) whether a given partial recursive function has a recursive domain, or (if we think it has a recursive domain) what the recursive characteristic function of the domain is. When saying that some function used in LISP is recursive (or converges for certain arguments), one
should be prepared to give a proof which amounts to more than hand-waving. Our formal system provides both medium and rationale for such proofs (when they exist).

In chapter V a mechanical procedure for finding characteristic functions is described. This procedure was suggested by the work in chapter IV on potential recursiveness, and can be applied with good results to a large class of functions. We feel that the program which was written to implement the procedure could form the nucleus of a new LISP debugging tool.

As a last introductory note, we mention the obvious limitation of our formal system: there is a recursive function which cannot be proved to be recursive in this system. This result follows immediately from the fact that the set of functions which can be proved to be recursive is a recursively enumerable set, while the set of all recursive functions is not recursively enumerable. Similarly, there is a potentially recursive function which cannot be proved to be potentially recursive in this system. A full treatment of a general theory of provable recursive functions (for a broad class of logical systems) may be found in Fischer. [1]
Chapter II
THE FORMAL SYSTEM

In this chapter we describe the language, interpretation, axioms and rules of inference of the formal system. We also present a Deduction Theorem for this system and an algol-like notation for proofs involving use of the Deduction Theorem.

The following is an extended-Backus-notation definition of the language of the formal system. (Extended-Backus-notation uses the following connectives: "<", ",", ";="", ";"|"", and "...". "<" and "," always enclose language elements. ";"|" means 'or'. ";="" means 'is'. "..." means 'any number (0,1,2,...) of'.)

\[
\langle \text{LETTER} \rangle ::= \text{A} | \text{B} | \text{C} | \ldots | \text{X} | \text{Y} | \text{Z} \\
\langle \text{letter} \rangle ::= \text{a} | \text{b} | \text{c} | \ldots | \text{x} | \text{y} | \text{z} \\
\langle \text{number} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \\
\langle \text{atomic symbol} \rangle ::= \langle \text{LETTER} \rangle \langle \text{atom part} \rangle \\
\langle \text{atom part} \rangle ::= \langle \text{empty} \rangle | \langle \text{LETTER} \rangle \langle \text{atom part} \rangle | \langle \text{number} \rangle \langle \text{atom part} \rangle \\
\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle \langle \text{id part} \rangle \\
\langle \text{id part} \rangle ::= \langle \text{empty} \rangle | \langle \text{letter} \rangle \langle \text{id part} \rangle | \langle \text{number} \rangle \langle \text{id part} \rangle \\
\langle \text{S-expr} \rangle ::= \langle \text{atomic symbol} \rangle | (\langle \text{S-expr} \rangle, \langle \text{S-expr} \rangle) \\
\quad \quad \quad (\langle \text{S-expr} \rangle \ldots \langle \text{S-expr} \rangle)
\]
\(<\text{constant}\> ::= <\text{S-expr}\>
\(<\text{variable}\> ::= <\text{identifier}\>
\(<\text{function}\> ::= <\text{identifier}\>
\(<\text{form}\> ::= <\text{constant}\> | <\text{variable}\> | <\text{function}\> [ <\text{form}\>; \cdots; <\text{form}\> ]
\quad \text{cond}[ <\text{form}\> \rightarrow <\text{form}\> ; \cdots; <\text{form}\> \rightarrow <\text{form}\> ]
\(<\text{term}\> ::= <\text{form}\>
\(<\text{atomic formula}\> ::= <\text{term}\> | <\text{term}\> = <\text{term}\> | * <\text{term}\>
\(<\text{function definition}\> ::= <\text{function}\> [ <\text{variable}\>; \cdots; <\text{variable}\> ] = \Delta <\text{form}\>
\(<\text{formula}\> ::= <\text{atomic formula}\> | \rightarrow <\text{formula}\> | ( <\text{formula}\> \rightarrow <\text{formula}\> )
\quad ( <\text{formula}\> \land <\text{formula}\> ) | ( <\text{formula}\> \lor <\text{formula}\> )
\quad ( <\text{formula}\> \leftrightarrow <\text{formula}\> )
\quad ( \forall <\text{variable}\> ) <\text{formula}\> | ( \exists <\text{variable}\> ) <\text{formula}\>
\(<\text{wff}\> ::= <\text{formula}\> | <\text{function definition}\>

In the above, S-expressions play the role of logical constants. Terms are both logical terms and logical predicate symbols. The atomic formula "*<\text{term}>" means "<\text{term}> is effectively calculable". A function definition serves to replace use of the LISP function label; it is not assigned an interpretation but it affects the interpretation of other wffs.

The logical connectives \(\rightarrow\) and \(\rightarrow\), and the quantifier \(\forall\), are the only symbols we will assign interpretations to. Formulas containing \(\land\), \(\lor\), \(\leftrightarrow\), and \(\exists\) will be considered to be abbreviations for equivalent formulas containing only \(\rightarrow\), \(\rightarrow\), and \(\forall\).
In particular,

\[ \phi \land \psi \equiv \neg (\phi \rightarrow \neg \psi) , \]

\[ \phi \lor \psi \equiv (\neg \phi \rightarrow \psi) , \]

\[ \phi \leftrightarrow \psi \equiv \neg ((\phi \rightarrow \psi) \rightarrow (\psi \rightarrow \phi)) , \]

and

\[ (\exists v) \phi \equiv \neg (\forall v) \neg \phi . \]

We will follow the usual conventions concerning elimination of parentheses. (In binding power, the connectives are ordered as follows: \( \leftrightarrow, \rightarrow, \forall, \exists, \lor, \land, \neg \).) Also, we follow the practice of weakening binary logical connectives by enclosing them within periods.

We define **free** and **bound** variable in the usual way. If \( A \) is a wff, \( t \) a term, and \( x \) a variable, \( t \) is said to be **free for \( x \) in \( A \)** if and only if no free occurrences of \( x \) in \( A \) lie within the scope of any quantifier \((\forall y)\), where \( y \) is a variable in \( t \).

Next we establish a standard interpretation of the language just defined.

First note that there is a simple lexicographical ordering of the syntactic class \( \langle \text{identifier} \rangle \), which establishes a 1:1 function from \( \omega \) (the set of all non-negative integers) onto the class \( \langle \text{identifier} \rangle \). We choose this ordering to be the standard ordering of the identifiers.

Let \( S \) be the set of all S-expressions. We assume that the notion of S-expression is previously defined and on the same intuitive level as the notion of integer. \( S^\omega \) denotes the set of all functions from \( \omega \) into \( S \).
$\vec{x} \in S^\omega$ should be thought of as an infinite sequence of $S$-expressions.

If $\vec{x} \in S^\omega$, $j \in \omega$, $q \in S$, then $\vec{x}(j \mid q)$ is the sequence obtained by replacing the $j^{th}$ element of $\vec{x}$ with $q$.

An interpretation $D$ is a triple $\langle A_D, P_D, f_D \rangle$, where

$A_D = S \cup \{\mu\}$ is the universe of the interpretation,

$P_D \subseteq \langle \text{function} \rangle$ is a set of function names, and

$f_D$ is a function with domain $P_D$; and for $g \in P_D$, $f_D(g)$ is a set of instructions for computing $g[\ldots \text{args} \ldots]$.

If $t$ is a term, and $\vec{x} \in S^\omega$, then we define $t_D(\vec{x})$, the interpretation of $t$ in $D$ given by $\vec{x}$, as follows:

Case I: $t$ is a constant. But $\langle \text{constant} \rangle = \langle S\text{-expr} \rangle = S$. So let $t_D(\vec{x}) = t$.

Case II: $t$ is a variable. Let $i$ be the index of $t$ in the standard ordering of the identifiers, and let $t_D(\vec{x}) = x_i$.

Case III: $t$ is of the form $g[t^1;\ldots;t^n]$.

Case 1: for some $i$, $t^i_D(\vec{x}) = \mu$. Then let $t_D(\vec{x}) = \mu$.

Case 2: $g \notin P_D$. Then let $t_D(\vec{x}) = \mu$.

But if $g \in P_D$, $f_D(g)$ gives instructions for computing $g$.

Let $C$ denote the computation of $g[t^1_D(\vec{x});\ldots;t^n_D(\vec{x})]$.

Case 3: $C$ does not converge. Then let $t_D(\vec{x}) = \mu$.

Case 4: $C$ converges and gives as output the $S$-expression $q$.

Then let $t_D(\vec{x}) = q$. 
Case IV: \( t \) is of the form \( \text{cond}[p^1 \rightarrow e^1; \ldots; p^n \rightarrow e^n] \).

Case 1: \( n = 0 \), \( t = \text{cond[]} \). Then let \( S_D(\bar{x}) = \mu \).

Case 2: \( p^1_D(\bar{x}) = \mu \). Then let \( S_D(\bar{x}) = \mu \).

Case 3: \( p^1_D(\bar{x}) \neq F \). Then let \( S_D(\bar{x}) = e^1_D(\bar{x}) \).

Case 4: \( p^1_D(\bar{x}) = F \). Then let \( S_D(\bar{x}) = (\text{cond}[p^2 \rightarrow e^2; \ldots; p^n \rightarrow e^n])D(\bar{x}) \).

Having defined the interpretation of terms, we proceed to define the meanings an interpretation can give to a formula. The interpretation \( D \) defines a partition of \( \langle \text{formula} \rangle \) for every \( \bar{x} \in S^\omega \). The equivalence classes of the partition are called \( T_D(\bar{x}) \), \( F_D(\bar{x}) \), and \( U_D(\bar{x}) \); and contain formulas which are \text{true}, \text{false}, and \text{undefined}, respectively, given \( \bar{x} \).

These categories are roughly equivalent to the truth values in Kleene's 3-valued logic.\(^2\)

Let \( \varphi \) be a formula, \( \bar{x} \in S^\omega \). The interpretation of \( \varphi \), given \( \bar{x} \), is determined as follows:

Case I: \( \varphi \) is an atomic formula.

Case 1: \( \varphi \) is a term.

Case a: \( \varphi D(\bar{x}) = \mu \). Then \( \varphi \in U_D(\bar{x}) \).

Case b: \( \varphi D(\bar{x}) \neq F \). Then \( \varphi \in T_D(\bar{x}) \).

Case c: \( \varphi D(\bar{x}) = F \). Then \( \varphi \in F_D(\bar{x}) \).

Case 2: \( \varphi \) is \( t^1 = t^2 \).

Case a: either \( t^1_D(\bar{x}) = \mu \) or \( t^2_D(\bar{x}) = \mu \). Then \( \varphi \in U_D(\bar{x}) \).

Case b: \( t^1_D(\bar{x}) = t^2_D(\bar{x}) \). Then \( \varphi \in T_D(\bar{x}) \).

Case c: \( t^1_D(\bar{x}) \neq t^2_D(\bar{x}) \). Then \( \varphi \in F_D(\bar{x}) \).
Case 3: \( \phi \) is \(*t\).

Case a: \( t_D(\overline{x}) = \mu \). Then \( \phi \in F_D(\overline{x}) \).

Case b: \( t_D(\overline{x}) \neq \mu \). Then \( \phi \in T_D(\overline{x}) \).

Case II: \( \phi \) is \( \neg \psi \).

Case 1: \( \psi \in T_D(\overline{x}) \). Then \( \phi \in F_D(\overline{x}) \).

Case 2: \( \psi \in F_D(\overline{x}) \). Then \( \phi \in T_D(\overline{x}) \).

Case 3: \( \psi \in U_D(\overline{x}) \). Then \( \phi \in U_D(\overline{x}) \).

Case III: \( \phi \) is \( \theta \rightarrow \psi \).

Case 1: \( \theta \in F_D(\overline{x}) \) or \( \psi \in T_D(\overline{x}) \). Then \( \phi \in T_D(\overline{x}) \).

Case 2: \( \theta \in T_D(\overline{x}) \) and \( \psi \in F_D(\overline{x}) \). Then \( \phi \in F_D(\overline{x}) \).

Case 3: otherwise. Then \( \phi \in U_D(\overline{x}) \).

Case IV: \( \phi \) is \( \forall x \psi \). Let \( i \) be the index of \( x \) in the standard ordering of the identifiers.

Case 1: for some \( q \in S \), \( \psi \in F_D(\overline{x}(i|q)) \). Then \( \phi \in F_D(\overline{x}) \).

Case 2: for all \( q \in S \), \( \psi \in T_D(\overline{x}(i|q)) \). Then \( \phi \in T_D(\overline{x}) \).

Case 3: otherwise. Then \( \phi \in U_D(\overline{x}) \).

When \( \phi \in T_D(\overline{x}) \), we say \( D \) satisfies \( \phi \) given \( \overline{x} \) and write \( D \models \phi[\overline{x}] \).

When \( \phi \notin T_D(\overline{x}) \), we write \( D \not\models \phi[\overline{x}] \). Let \( \sigma \) be a sentence (i.e., a formula with no free variables). If \( D \models \sigma[\overline{x}] \) for some \( \overline{x} \), then \( B \models \sigma[\overline{x}] \) for all \( \overline{x} \) and we write \( D \models \sigma \). The same is true for \( \not\models \).
A standard interpretation $D = \langle A_D, P_D, f_D \rangle$ is an interpretation for which:

1. $\{\text{car, cdr, cons, atom, eq} \} \subseteq P_D$
2. $D$ satisfies all of the following sentences:

   - $(\forall x)(\forall y) \rightarrow \text{atom[cons}[x; y]]$
   - $(\forall x)(\forall y) \text{car[cons}[x; y]] = x$
   - $(\forall x)(\forall y) \text{cdr[cons}[x; y]] = y$
   - $(\forall x)(\rightarrow \text{atom}[x] \leftrightarrow \text{cons}[\text{car}[x]; \text{cdr}[x]] = x)$
   - $(\forall x) \ast \text{atom}[x]$
   - $(\forall x)(\rightarrow \text{atom}[x] \rightarrow \ast \text{car}[x] \land \ast \text{cdr}[x])$
   - $(\forall x)(\forall y) \ast \text{cons}[x; y]$
   - $(\forall x)(\forall y) (\text{atom}[x] \land \text{atom}[y] \rightarrow \ast \text{eq}[x; y] \land (\text{eq}[x; y] \leftrightarrow x = y))$.

Although the model theory of LISP (which we have just established the framework of) is of some interest, it is not our major goal. We are more concerned with a proof theory of LISP: hence we shall continue to develop our axiomatic theory. The next step is to define axioms and rules of inference for the system.

The axioms of the formal system are as follows:

**LOGICAL AXIOMS**

I: $(\text{Def}(\varphi_1) \land \ldots \land \text{Def}(\varphi_n) \rightarrow \tau(\varphi_1, \ldots, \varphi_n))$, for any wffs $\varphi_1, \ldots, \varphi_n$ and all tautologies $\tau$, where $\tau(\varphi_1, \ldots, \varphi_n)$ is an instance of the tautology $\tau$ and $\text{Def}(\varphi)$ is a wff, defined as follows:
Case I:  $\varphi$ is a term. Then $\text{Def}(\varphi) = \star \varphi$.

Case II:  $\varphi$ is $t_1 = t_2$. The $\text{Def}(\varphi) = (\star t_1 \wedge \star t_2)$.

Case III:  $\varphi$ is $\star t$. The $\text{Def}(\varphi) = T$.

Case IV:  $\varphi$ is $\neg \psi$. Then $\text{Def}(\varphi) = \text{Def}(\psi)$.

Case V:  $\varphi$ is $\theta \rightarrow \psi$. Then $\text{Def}(\varphi) =$

$$\text{(Def}(\theta) \wedge \text{Def}(\psi)) \vee (\psi \wedge \text{Def}(\psi)) \vee (\neg \theta \wedge \text{Def}(\theta))$$

Case VI:  $\varphi$ is $(\forall \psi) \psi$. Then $\text{Def}(\varphi) =$

$$((\forall \psi) \text{Def}(\psi)) \wedge (\forall \psi) (\neg \psi \wedge \text{Def}(\psi))$$

II:  $(\star t \rightarrow ((\forall x) \alpha(x) \rightarrow \alpha(t)))$, for all wffs $\alpha(x)$ and all terms $t$
where $t$ is free for $x$ in $\alpha(x)$.

III:  $(\forall x)(\alpha \rightarrow (\alpha \rightarrow (\forall x) \alpha))$, for all wffs $\alpha$ and $\alpha$
where $\alpha$ contains no free occurrences of $x$.

PROPER AXIOMS

I: equality

$(\forall x)(x = x)$

$(\forall x)(\forall y)(x = y \rightarrow y = x)$

$(\forall x)(\forall y)(\forall z)(x = y \wedge y = z \rightarrow x = z)$

$(\forall x)(\forall y)(x = y \rightarrow t(x) = t(y))$, where $t(y)$ is a term with some
occurrences of $x$ replaced by $y$.

$(\forall x)(\forall y)(x = y \rightarrow (\alpha(x) \rightarrow \alpha(y)))$, where $\alpha(y)$ is a wff with some
occurrences of $x$ replaced by $y$.
II: arithmetic structure

\( \forall x \forall y \rightarrow \text{atom[cons}[x; y]] \)
\( \forall x \forall y \text{ car[cons}[x; y]] = x \)
\( \forall x \forall y \text{ cdr[cons}[x; y]] = y \)
\( \forall x \rightarrow \text{atom}[x] \rightarrow \text{cons}[\text{car}[x]; \text{cdr}[x]] = x \)

III: induction

\( \left( \forall x \left( \text{atom}[x] \rightarrow A(x) \right) \right) \)
\( \land \left( \forall x \forall y \left( A(x) \land A(y) \rightarrow A(\text{cons}[x; y]) \right) \right) \)
\( \rightarrow \left( \forall x \ A(x), \text{ for all wffs } A \right) \)

IV: convergences

\( \forall x \ * \text{atom}[x] \)
\( \forall x \left( \neg \text{atom}[x] \rightarrow * \text{car}[x] \land * \text{cdr}[x] \right) \)
\( \forall x \forall y \ * \text{cons}[x; y] \)
\( \forall x \forall y \left( \text{atom}[x] \land \text{atom}[y] \right) \)
\( \rightarrow * \text{eq}[x; y] \land (\text{eq}[x; y] \leftrightarrow x = y) \)

\(*c, \text{ for all constants } c\)
\(*v, \text{ for all variables } v\)

V: computation

\( \rightarrow \ * \text{cond}[\] \)
\( \rightarrow * p \rightarrow * \text{cond}[p \rightarrow e; p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n], \text{ where} \)
\( p \text{ and } e \text{ are forms, } p_i \text{ and } e_i \text{ are forms } (1 \leq i \leq n) \text{ and } n \geq 0 \)
cond[q→e; p₁→e₁;...;pₙ→eₙ] = e, where q is any constant except F and pᵢ and eᵢ are forms (1≤i≤n) and e is a form and n≥0

cond[F→e; p₁→e₁;...; pₙ→eₙ] = cond[p₁→e₁;...;pₙ→eₙ],

where pᵢ and eᵢ are forms (1≤i≤n) and e is a form and n≥0

*f[t₁;...;tₙ]→*t₁, where 1≤i≤n and f is a function and the tᵢ are forms (1≤k≤n) and n≥1

atom[A], for all atomic symbols A

cons[X;Y] = (X,Y), for all S-expr's X and Y

(S₁ S₂...Sₙ) = (S₁.(S₂.(...((Sₙ.NIL)...))), for any S-expr's S₁,...,Sₙ and all n≥0

(p = F)→¬p, for all terms p

The rules of inference are:

I: modus ponens (MP) \[
\frac{\ a \ a}{b}
\]

II: generalization (Gen) \[
\frac{\ a}{(\forall x)\ a}
\]

A deduction is a finite sequence of wffs of which each is an axiom, or is derived from one or two of the preceding wffs in the sequence by application of one of the rules of inference. A proof of a wff A is a deduction whose terminal wff is A. When a proof of A exists, we
write \( \Gamma \vdash \bar{a} \). A deduction from hypotheses \( \Gamma \), where \( \Gamma \) is a set of wffs, is a deduction in which members of \( \Gamma \) play the role of additional axioms. A proof from hypotheses \( \Gamma \) of \( \bar{a} \) is a deduction from hypotheses \( \Gamma \) whose terminal wff is \( \bar{a} \). When a proof of \( \bar{a} \) from hypotheses \( \Gamma \) exists, we write \( \Gamma \vdash \bar{a} \).

Let \( \bar{a} \in \Gamma \), and let \( B_1, \ldots, B_n \) be a deduction with hypotheses \( \Gamma \). We say that \( B_i \) depends on \( \bar{a} \) when

1. \( B_i \) is \( \bar{a} \) and the justification is that \( B_i \in \Gamma \), or
2. \( B_i \) is a direct consequence of one or two of the preceding wffs, at least one of which depends on \( \bar{a} \).

**Deduction Theorem:** Assume that \( \Gamma, \bar{a} \vdash \bar{b} \), where, in the deduction, no application of Gen to a formula which depends on \( \bar{a} \) has as its quantified variable a variable free in \( \bar{a} \). Then \( \Gamma \vdash \bar{a} \rightarrow \bar{b} \).

**Proof:** see e.g. Mendelson\(^{[3]}\).

**Corollary:** If a deduction \( \Gamma, \bar{a} \vdash \bar{b} \) involves no application of Gen when the quantified variable is free in \( \bar{a} \), then \( \Gamma \vdash \bar{a} \rightarrow \bar{b} \).

**Corollary:** If \( \bar{a} \) is closed, and \( \Gamma, \bar{a} \vdash \bar{b} \), then \( \Gamma \vdash \bar{a} \rightarrow \bar{b} \).

Rather than specify deductions in complete detail, we will adopt a shorthand notation which will allow syntactic use of the deduction theorem. If we wish to prove \( (A \rightarrow (B \rightarrow C)) \), the form of the proof would be:
begin assume A
  begin assume B
  \{ \}
  \{ \}
  \text{the deduction } \{A, B\} \vdash C
  \text{then } C \text{ end}
  \text{then } (B \rightarrow C) \text{ end}
(A \rightarrow (B \rightarrow C))

We give a complete syntactic specification of this proof language as follows:

\[
\begin{align*}
\langle \text{proof} \rangle & ::= \langle \text{deflist} \rangle \langle \text{proof end} \rangle \\
\langle \text{deflist} \rangle & ::= \langle \text{empty} \rangle \mid \langle \text{function definition} \rangle \langle \text{deflist} \rangle \\
\langle \text{proof end} \rangle & ::= \langle \text{empty} \rangle \mid \langle \text{proof step} \rangle \langle \text{proof end} \rangle \\
\langle \text{proof step} \rangle & ::= \langle \text{formula} \rangle \mid \langle \text{deduction} \rangle \\
\langle \text{deduction} \rangle & ::= \text{begin assume } \langle \text{formula} \rangle \langle \text{deduction end} \rangle \\
\langle \text{deduction end} \rangle & ::= \text{then } \langle \text{formula} \rangle \text{ end } \langle \text{proof step} \rangle \langle \text{deduction end} \rangle
\end{align*}
\]

A valid proof in this language is any proof which meets the following requirements:

(1) Within a begin...end pair, all formulas, except those immediately preceded by another begin...end pair, must be axioms, or must be immediate consequences of preceding formulas in this begin...end pair, or an outer begin...end pair, by application of MP
or Gen, except no application of Gen to a formula which depends on an assumption may have as its quantified variable a variable free in the assumption.

(2) Every begin...end pair must be followed immediately by a formula or a then clause, and the formula (in either case) must be:

(assumption→conclusion).

(3) If not within a begin...end pair, formulas must be axioms, or covered by requirement (2), or immediate consequences of preceding formulas (which must also not be within any begin...end pair) by application of MP or Gen.
Chapter III

SOME FORMAL PROOFS

In this chapter we will use our formal system to define a few fairly well-known LISP functions and prove a few fairly well-known facts about them. These proofs are included not so much for their content as for their value as examples of thoroughly formal proofs.

First we investigate a few minor functions not provided by the standard axioms.

\[
\text{not}[x] =_A \text{cond}[x \rightarrow F ; T \rightarrow T].
\]

\[
\text{begin assume } \neg x = F
\]

\[
\text{not}[x] = \text{cond}[x \rightarrow F ; T \rightarrow T] = F
\]

\[
\text{*F}
\]

\[
\text{then } \neg \text{not}[x] \text{ end}
\]

\[
\neg x = F \rightarrow \neg \text{not}[x]
\]

\[
\text{begin assume } x = F
\]

\[
\text{not}[x] = \text{cond}[x \rightarrow F ; T \rightarrow T]
\]

\[
= \text{cond}[T \rightarrow T] = T
\]

\[
\text{*T}
\]

\[
\text{then } \neg \text{not}[x] \text{ end}
\]

\[
x = F \rightarrow \neg \text{not}[x]
\]

\[
\neg \text{not}[x]
\]

\[
(\forall x) \neg \text{not}[x]
\]
and \([x;y] = \Delta \text{cond}[not[x] \rightarrow F; T \rightarrow y]\)

\[\begin{align*}
\text{begin assume } & \neg x = F \\
\text{and } [x;y] & = \text{cond}[not[x] \rightarrow F; T \rightarrow y] \\
& = \text{cond}[T \rightarrow y] = y \\
* y \\
\text{then } & * \text{and}[x;y] \text{ end} \\
\neg x & = F \leftrightarrow * \text{and}[x;y]
\end{align*}\]

\[\begin{align*}
\text{begin assume } & x = F \\
\text{and } [x;y] & = \text{cond}[not[x] \rightarrow F; T \rightarrow y] \\
& = F \\
* F \\
\text{then } & * \text{and}[x;y] \text{ end} \\
x & = F \leftrightarrow * \text{and}[x;y]
\end{align*}\]

\[\forall x \forall y (* \text{and}[x;y])\]

or \([x;y] = \Delta \text{cond}[x \rightarrow T; T \rightarrow y]\)

\[\begin{align*}
\text{begin assume } & \neg x = F \\
\text{or } [x;y] & = \text{cond}[x \rightarrow T; T \rightarrow y] \\
& = T \\
* T \\
\text{then } & * \text{or}[x;y] \text{ end} \\
\neg x & = F \leftrightarrow * \text{or}[x;y]
\end{align*}\]

\[\begin{align*}
\text{begin assume } & x = F \\
\text{or } [x;y] & = \text{cond}[x \rightarrow T; T \rightarrow y] \\
& = \text{cond}[T \rightarrow y] = y \\
* y \\
\text{def}
\end{align*}\]
\begin{align*}
\text{then } \star \text{or}[x;y] \text{ end} \\
x = F \leftrightarrow \star \text{or}[x;y] \\
\star \text{or}[x;y] \\
(\forall x)(\forall y)(\star \text{or}[x;y]) \\
\text{null}[x] = \triangle \text{cond}[\text{not}[\text{atom}[x]] \rightarrow F; T \rightarrow \text{eq}[x;\text{NIL}]] \\
\text{begin assume } \text{atom}[x] \\
\quad \text{null}[x] = \text{cond}[\text{not}[\text{atom}[x]] \rightarrow F; T \rightarrow \text{eq}[x;\text{NIL}]] \\
\quad \quad = \text{eq}[x;\text{NIL}] \\
\quad \text{atom}[\text{NIL}] \\
\quad \text{*eq}[x;\text{NIL}] \\
\quad \text{then } \star \text{null}[x] \text{ end} \\
\text{atom}[x] \leftrightarrow \star \text{null}[x] \\
\text{begin assume } \rightarrow \text{atom}[x] \\
\quad \text{null}[x] = \text{cond}[\text{not}[\text{atom}[x]] \rightarrow F; \text{etc}] \\
\quad \quad = F \\
\quad \star F \\
\quad \text{then } \star \text{null}[x] \text{ end} \\
\rightarrow \text{atom}[x] \leftrightarrow \star \text{null}[x] \\
\star \text{null}[x] \\
(\forall x)\star \text{null}[x] \\
\text{begin assume } \text{null}[x] \\
\quad \text{begin assume } \rightarrow \text{atom}[x] \\
\quad \quad \text{null}[x] = \text{cond}[\text{not}[\text{atom}[x]] \rightarrow F; \text{etc}] \\
\quad \quad \quad = F \\
\quad \quad \text{then } \rightarrow \text{null}[x] \text{ end}
\end{align*}

Ax + MP

\begin{align*}
taut + \text{conj} + \text{MP} \\
\text{Gen} + \text{Gen}
\end{align*}
\[ \neg \text{atom}[x] \iff \neg \text{null}[x] \]
\[ \text{then atom}[x] \text{ end} \]
\[ \text{null}[x] \iff \text{atom}[x] \]
\[ (\forall x)(\text{null}[x] \iff \text{atom}[x]) \]

To summarize, we have shown
\[ \vdash (\forall x)\neg \text{not}[x] \]
\[ \vdash (\forall x)(\forall y)\neg \text{and}[x;y] \]
\[ \vdash (\forall x)(\forall y)\neg \text{or}[x;y] \]
\[ \vdash (\forall x)\neg \text{null}[x] \]
\[ \vdash (\forall x)(\text{null}[x] \iff \text{atom}[x]) \]

Next we prove a sentence which we originally thought would be an axiom.

\[ \text{finite}[x] = \Delta \text{cond}[\text{atom}[x] \iff \text{T}; \text{T} \iff \text{and}[\text{finite}[\text{car}[x]]; \text{finite}[\text{cdr}[x]]]) \]
\[ \begin{align*}
\text{begin assume } & \text{atom}[x] \\
& \text{finite}[x] = \text{cond}[\text{atom}[x] \iff \text{T}; \text{etc}] \\
& \quad = \text{T} \\
& \ast \text{T} \\
\text{then } & \ast \text{finite}[x] \text{ end} \\
\text{atom}[x] & \iff \ast \text{finite}[x] \\
(\forall x)(\text{atom}[x] & \iff \ast \text{finite}[x]) \\
\text{begin assume } & \ast \text{finite}[x] \land \ast \text{finite}[y] \\
& \text{finite}[\text{cons}[x;y]] = \text{cond}[\text{atom}[\text{cons}[x;y]] \iff \text{T}; \\
& \quad \text{T} \iff \text{and}[\text{finite}[\text{car}[\text{cons}[x;y]]]; \\
& \quad \text{finite}[\text{cdr}[\text{cons}[x;y]]])] \\
\end{align*} \]
\[\text{atom[cons[x;y]]}\]
\[\text{finite[cons[x;y]]} = \text{and[finite[x];finite[y]]}\]
\[(\forall x)(\forall y)\text{*and[x;y]}\]
\[\text{*and[finite[x];finite[y]]}\]
\[\text{then *finite[cons[x;y]] end}\]
\[\text{*finite[x] } \land \text{ *finite[y]} \rightarrow *\text{finite[cons[x;y]]}\]
\[(\forall x)(\forall y)(\text{*finite[x] } \land \text{ *finite[y]} \rightarrow *\text{finite[cons[x;y]]})\]
\[(\forall x)(\text{*finite[x]})\]
by induction

It would appear that induction is a much stronger principle than
\[(\forall x)(\text{*finite[x]})\]. In all of the following proofs, we rely mainly on induction.

Next, we tackle a few not-so-trivial proofs. We will be concerned
with \text{listp}, a predicate true of lists, and the function \text{append}, which
(supposedly) appends two lists. These functions are defined as follows:
\[\text{listp[x]} = \Delta \text{cond[null[x] } \rightarrow \text{T;atom[x] } \rightarrow \text{F;T } \rightarrow \text{listp[cdr[x]]}}\]
\[\text{append[x;y]} = \Delta \text{cond[null[x] } \rightarrow \text{y;T } \rightarrow \text{cons[car[x];append[cdr[x];y]]}}\].

We will demonstrate the following:
\[\vdash (\forall x) *\text{listp[x]}\]
\[\vdash (\forall x)(\forall y)(\text{listp[x]} \rightarrow *\text{append[x;y]})\]
\[\vdash (\forall x)(\forall y)(\text{listp[x]} \land \text{listp[y]} \rightarrow *\text{listp[append[x;y]]})\].

\begin{verbatim}
begin assume \text{atom[x]}

begin assume \text{null[x]}

\text{listp[x]} = \text{cond[null[x] } \rightarrow \text{T;etc]}

= \text{T}

\text{then *listp[x] end}
\end{verbatim}
null[x] ⇒ *listp[x]

begin assume null[x]

listp[x] = cond null[x] ⇒ T; atom[x] ⇒ F; etc
  = cond atom[x] ⇒ F; etc
  = F

then *listp[x] end

⇒ null[x] ⇒ *listp[x]

then *listp[x] end

atom[x] ⇒ *listp[x]

(∀x)(atom[x] ⇒ *listp[x])

begin assume *listp[p] ∧ *listp[q]

⇒ null[cons[p;q]]

listp[cons[p;q]] = cond null[cons[p;q]] ⇒ T; atom[cons[p;q]] ⇒ F;
  ⇒ listp[cdr[cons[p;q]]]
  = listp[q]

*listp[p] ∧ *listp[q] ⇒ *listp[cons[p;q]]

(∀p)(∀q)(*listp[p] ∧ *listp[q] ⇒ *listp[cons[p;q]])

(∀x)(*listp[x])

begin assume atom[x]

begin assume listp[x]

begin assume null[x]

append[x;y] = cond null[x] ⇒ y; etc
  = y

then *append[x;y] end
null[x] $\rightarrow$ *append [x;y]

begin assume $\neg$ null[x]

listp[x] = cond [null[x] $\rightarrow$ T; atom[x] $\rightarrow$ F; etc] 

= F

$\neg$ listp[x]

F

then *append [x;y] end

$\neg$ null[x] $\rightarrow$ *append [x;y]

then *append [x;y] end

then listp[x] $\rightarrow$ *append [x;y] end

atom[x] $\rightarrow$ (listp[x] $\rightarrow$ *append [x;y])

begin assume (listp[p] $\rightarrow$ *append [p;y]) $\wedge$ (listp[q] $\rightarrow$ *append [q;y])

begin assume listp[cons[p;q]]

$\neg$ null[cons[p;q]]

listp[cons[p;q]] = cond [null[cons[p;q]] $\rightarrow$ T; atom[cons[p;q]] $\rightarrow$ F;

T $\rightarrow$ listp[cdr[cons[p;q]]]]

= listp[q]

listp[q]

*append [q;y]

append[cons[p;q];y] = cond [null[cons[p;q]] $\rightarrow$ y;

T $\rightarrow$ cons[car[cons[p;q]]];

append[cdr[cons[p;q]];y]]

= cons[p;append[q;y]]

($\forall x$)($\forall y$) *cons [x;y]

*cons [p;append[q;y]]

then *append [cons [p;q];y] end
then listp[cons[p;q]] \rightarrow*append[cons[p;q];y] end

(\forall p)(\forall q)((listp[p] \rightarrow*append[p;y]) \land (listp[q] \rightarrow*append[q;y]))

\rightarrow. (listp[cons[p;q]] \rightarrow*append[cons[p;q];y])

(\forall x)(\forall y)(listp[x] \rightarrow*append[x;y])

begin assume atom[x]

begin assume listp[x] \land listp[y]

begin assume null[x]

listp[x] = \text{cond}[null[x] \rightarrow T; atom[x] \rightarrow F; etc]

= F

then F end

\rightarrow null[x] \rightarrow F

null[x]

append[x;y] = \text{cond}[null[x] \rightarrow y; etc]

= y

listp[append[x;y]] = listp[y]

then listp[append[x;y]] end

then listp[x] \land listp[y] \rightarrow listp[append[x;y]] end

(\forall x)(atom[x] \rightarrow (listp[x] \land listp[y] \rightarrow listp[append[x;y]])

begin assume (listp[p] \land listp[y] \rightarrow listp[append[p;y]])

\land (listp[q] \land listp[y] \rightarrow listp[append[q;y]])

\rightarrow null[cons[p;q]]

append[cons[p;q];y] = \text{cond}[null[cons[p;q]] \rightarrow y;]

T \rightarrow cons[car[cons[p;q]]];

appeend[cdr[cons[p;q]];y]]

= cons[p;append[q;y]]
\begin{verbatim}
begin assume listp[cons[p;q]] \land listp[y]

listp[append[cons[p;q];y]] = listp[cons[p;append[q;y]]]

listp[cons[p;q]] = cond [null[cons[p;q]] \rightarrow T; atom[cons[p;q]] \rightarrow F;
                      T \rightarrow listp[cdr[cons[p;q]]]]

= listp[q]

listp[q]

listp[append[q;y]]

listp[append[cons[p;q];y]] = listp[cons[p;append[q;y]]]

= cond [null[cons[p;append[q;y]]] \rightarrow T;
        atom[cons[p;append[q;y]]] \rightarrow F;
        T \rightarrow listp[cdr[cons[p;append[q;y]]]]]

= listp[append[q;y]]

\textbf{then} listp[append[cons[p;q];y]] end

\textbf{then} listp[cons[p;q]] \land listp[y] \rightarrow listp[append[cons[p;q];y]] end

(\forall p)(\forall q)((listp[p] \land listp[y] \rightarrow listp[append[p;y]])

\land (listp[q] \land listp[y] \rightarrow listp[append[q;y]]))

\rightarrow (listp[cons[p;q]] \land listp[y] \rightarrow listp[append[cons[p;q];y]])

(\forall x)(listp[x] \land listp[y] \rightarrow listp[append[x;y]])

(\forall x)(\forall y)(listp[x] \land listp[y] \rightarrow listp[append[x;y]])

Another fairly important function is equal.

equal[x;y] = cond[atom[x] \rightarrow cond[atom[y] \rightarrow eq[x;y]; T \rightarrow F];

atom[y] \rightarrow F;

T \rightarrow and[equal[car[x];car[y]]; equal[cdr[x];cdr[y]]]

The following two theorems are of interest:

\vdash (\forall x)(\forall y) equal[x;y]

\vdash (\forall x)(\forall y) (equal[x;y] \leftrightarrow x = y)
\end{verbatim}
Finally, we wish to show that any function defined by primitive recursion preserves recursiveness. We say that a function \( g[x;y] \) is defined primitive recursively in terms of \( s[x;y] \), \( k[x;y;a;b] \), \( m[y] \), and \( n[y] \) when

\[
g[x;y] = \begin{cases} \text{cond} \left[ \text{atom}[x] \rightarrow s[x;y]; \right. \\ T \rightarrow k[x;y;g[\text{car}[x];m[y]];g[\text{cdr}[x];n[y]]] \end{cases}.
\]

We will show that

\[
\left\{ (\forall x)(\forall y)s[x;y], \\
(\forall x)(\forall y)(\forall a)(\forall b)k[x;y;a;b], \\
(\forall y)m[y], (\forall y)n[y] \right\} \vdash (\forall x)(\forall y)g[x;y].
\]

\textbf{begin assume} atom[x]
\begin{align*}
g[x;y] &= \text{cond} \left[ \text{atom}[x] \rightarrow s[x;y]; \text{etc} \right] \\
&= s[x;y] \\
(\forall x)(\forall y)s[x;y] \\
\text{then} \quad (\forall y)g[x;y] \quad \text{end}
\end{align*}

\( (\forall x)(\text{atom}[x] \rightarrow (\forall y)g[x;y]) \)

\textbf{begin assume} (\forall y)g[p;y] \land (\forall y)g[q;y]
\begin{align*}
g[\text{cons}[p;q];y] &= \text{cond} \left[ \text{atom}[\text{cons}[p;q]] \rightarrow s[\text{cons}[p;q];y]; \\
&= k[\text{cons}[p;q];y;g[\text{car}[\text{cons}[p;q]};m[y]]; \\
g[\text{cdr}[\text{cons}[p;q]];n[y]]] \\
(\forall y)g[p;y] \\
(\forall y)g[q;y] \\
g[p;m[y]] \\
(\forall y)g[q;n[y]] \\
\text{cons}[p;q] \\
\end{align*}
\*k[\text{cons}[p;q];y;g[p;m[y]];g[q;n[y]]]
\*g[\text{cons}[p;q];y]
\text{then } \forall(y)\ast g[\text{cons}[p;q];y] \text{ end }
(\forall(p))(\forall(q))(\forall(y)\ast g[p;y] \land (\forall(y)\ast g[q;y]) \rightarrow (\forall(y)\ast g[\text{cons}[p;q];y]))
(\forall(x))(\forall(y)\ast g[x;y])

In addition, any function defined by k-recursion (i.e., k-fold completely nested recursion) preserves recursiveness. It is fairly difficult to define k-recursion in closed form, but we present the normal form \cite{4} for 2-recursive definitions:

\(g[x;y;z] = \begin{cases} \text{cond} & [\text{or}[\text{atom}[x];\text{atom}[y]] \rightarrow s[x;y;z]; \\
T \rightarrow k[x;y;z]; \\
g[\text{car}[x];m_1[x;y;z];g[x;\text{car}[y];n_1[z]]]; \\
g[x;\text{cdr}[y];n_2[z]];n_3[z]); \\
g[\text{cdr}[x];m_2[x;y;z];g[x;\text{car}[y];n_4[z]]]; \\
g[x;\text{cdr}[y];n_5[z]];n_6[z]]) \end{cases}\)

The proof that 2-recursion (like 1- or primitive recursion, above) preserves recursiveness, is left to the reader. (Hint: Induction is required only k times; i.e., twice for this schema.)
Chapter IV

POTENTIAL RECURSIVENESS

In this chapter we will develop a small theory of potential recursive\[^5\] functions. A potential recursive function, roughly speaking, is a partial recursive function which can be made total recursive (has a total recursive extension). Formally, we define \( f \) to be a potential recursive function when \( f \) is a partial recursive function, and \( f \) has a recursive domain. When \( g \) is a recursive characteristic function for the domain of \( f \), we will say that \( g \) is a characteristic function for \( f \), or a charf for \( f \).

Potential recursive functions are of interest in the context of this thesis because all the basic LISP functions are potential recursive at least, and potential recursiveness is preserved under substitution, conditional computation, and \( k \)-recursion. However, we do not develop a theory of potential recursive functions beyond proofs of these preservation properties, since these are the only properties required to give theoretical justification to the program described in chapter V.

Another topic of interest to us is the area of proofs of potential recursiveness within the formal system already established. To prove that \( g \) is a charf of \( f \), it is necessary to demonstrate \( (\forall x) \cdot g[x] \) and \( (\forall x)(g[x] \iff f[x]) \). We will use our formal system to establish the preservation theorems below.
First, for completeness, we mention again that the basic LISP functions are potential recursive. In particular, \texttt{cons[x;y]} and \texttt{atom[x]} are recursive, \texttt{car[x]} and \texttt{cdr[x]} converge when \texttt{not[atom[x]]}, and \texttt{eq[x;y]} converges when \texttt{and[atom[x]; atom[y]]}. (Although \texttt{and} is not a basic LISP function, it is previously defined and proved to be recursive.)

**Theorem:** If \( f[x] \) and \( g[x] \) are potential recursive functions with charfs \( p[x] \) and \( q[x] \), respectively, then \( f[g[x]] \) is potential recursive and has charf

\[
s[x] = \begin{cases} \text{cond[not[}q[x]]} & \rightarrow \text{F}; \ T \rightarrow p[g[x]] \end{cases}.
\]

**Proof:** We first show \( s[x] \) to be recursive. We may assume that \( p[x] \) and \( g[x] \) are recursive.

\[
(\forall x) * q[x] \\
(\forall x) * \text{not}[x] \\
(\forall x) * \text{not}[q[x]]
\]

**begin assume not[q[x]]**

\[
s[x] = \text{cond[not[}q[x]]} \rightarrow \text{F}; \ \text{etc}
\]

\[
= \text{F}
\]

**then \( s[x] \) end**

\[
\text{not[q[x]]} \rightarrow \text{*s[x]}
\]

**begin assume \( \neg \text{not[q[x]]} \)**

\[
q[x]
\]

\[
s[x] = \text{cond[not[}q[x]]} \rightarrow \text{F}; \ \text{T} \rightarrow p[g[x]]
\]

\[
= p[g[x]]
\]
\[ *g[x] \]
\[ (\forall x) \ *p[x] \]
\[ *p[g[x]] \]
\[ \text{then } *s[x] \text{ end} \]
\[ \neg\neg q[x] \rightarrow *s[x] \]
\[ (\forall x) \ *s[x] \]

Next, we demonstrate \( s[x] \leftrightarrow \neg f[g[x]] \).

```
begin assume s[x]

begin assume not[q[x]]

\[ s[x] = \text{cond}\{\not q[x]\} \rightarrow F; \text{etc} \]
\[ = F \]

\[ \text{then } F \text{ end} \]

\[ \not q[x] \rightarrow F \]

\[ \neg \neg q[x] \]

q[x]

\[ *g[x] \]

\[ s[x] = \text{cond}\{\not q[x]\} \rightarrow F; T \rightarrow p[g[x]] \]
\[ = p[g[x]] \]

p[g[x]]

\[ (\forall x) \ p[x] \leftrightarrow *f[x] \]

\[ p[g[x]] \leftrightarrow *f[g[x]] \]

\[ \text{then } *f[g[x]] \text{ end} \]

\[ s[x] \leftrightarrow *f[g[x]] \]
```
begin assume *f[g[x]]

*g[x]
p[g[x]]
q[x]

s[x] = cond[not[q[x]]→F; T→p[g[x]]]
   = p[g[x]]
   = T 

then s[x] end

* f[g[x]]→s[x]

(∀x)(s[x]↔*f[g[x]])

Theorem: If f[x] and g[x] are potential recursive functions with charfs p[x] and q[x], respectively, then cond[f[x]→g[x]] is potential recursive and has charf

s[x] = A cond[not[p[x]]→F; f[x]→q[x]; T→F].

Proof: We first show s[x] to be recursive. We may assume that p[x] and q[x] are recursive.

(∀x) *p[x]

(∀x) *not[x]

*not[p[x]]

begin assume not[p[x]]

s[x] = cond[not[p[x]]→F; etc]
   = F 

then s[x] end

not[p[x]]→*s[x]
\begin{align*}
\text{begin assume } & \neg p(x) \\
p(x) & \\
\neg f(x) & \\
\text{begin assume } & f(x) \\
s(x) & = \text{cond}(\neg p(x) \rightarrow F; f(x) \rightarrow q(x); \text{ etc}) \\
& = q(x) \\
(\forall x) & \neg q(x) \\
\text{then } & \neg s(x) \text{ end} \\
f(x) & \rightarrow \neg s(x) \\
\begin{align*}
\text{begin assume } & \neg f(x) \\
s(x) & = \text{cond}(\neg p(x) \rightarrow F; f(x) \rightarrow q(x); T \rightarrow F) \\
& = F \\
\text{then } & \neg s(x) \text{ end} \\
\neg f(x) & \rightarrow \neg s(x) \\
\text{then } & \neg s(x) \text{ end} \\
\neg \neg p(x) & \rightarrow \neg s(x) \\
(\forall x) & \neg s(x) \\
\end{align*}

Next, we demonstrate \( s(x) \leftrightarrow \neg \text{cond}(f(x) \rightarrow g(x)) \).

\begin{align*}
\text{begin assume } & s(x) \\
\text{begin assume } & \neg p(x) \\
\text{then } & F \text{ end} \\
p(x) & \\
\neg f(x) &
\end{align*}
begin assume $\neg f[x]$

$$s[x] = \text{cond}[\neg p[x]] \rightarrow F; \quad f[x] \rightarrow q[x]; \quad T \rightarrow F$$

$= F$

then $F$ end

$f[x]$

$s[x] = q[x]$

$q[x]$

$\text{cond}[f[x] \rightarrow g[x]] = g[x]$

$\neg g[x]$

then $\neg \text{cond}[f[x] \rightarrow g[x]]$ end

$s[x] \rightarrow \neg \text{cond}[f[x] \rightarrow g[x]]$

begin assume $\neg \text{cond}[f[x] \rightarrow g[x]]$

$\neg f[x]$

$p[x]$

begin assume $\neg f[x]$

$\text{cond}[f[x] \rightarrow g[x]] = \text{cond}[]$

$\rightarrow \neg \text{cond}[]$

$\rightarrow \neg \text{cond}[f[x] \rightarrow g[x]]$

then $F$ end

$f[x]$

$\text{cond}[f[x] \rightarrow g[x]] = g[x]$

$\neg g[x]$

$q[x]$
\[ s[x] = \text{cond(\not p[x])} \rightarrow F; \ f[x] \rightarrow q[x]; \ T \rightarrow F \]

\[ = q[x] \]

then \( s[x] \) end

\(*\text{cond(} f[x] \rightarrow g[x]) \rightarrow s[x] \)

\((\forall x)(s[x] \leftrightarrow *\text{cond(} f[x] \rightarrow g[x]))\)

**Theorem:** If \( f[x], \ g[x], \) and \( h[x] \) are potential recursive functions with charfs \( p[x], \ q[x], \) and \( r[x] \) respectively, then \( \text{cond(} f[x] \rightarrow g[x]; \ T \rightarrow h[x]) \)

is potential recursive and has charf

\[ s[x] = \Delta \text{cond(\not p[x])} \rightarrow F; \ f[x] \rightarrow q[x]; \ T \rightarrow r[x] \]

**Proof:** First we show \( s[x] \) to be recursive. We may assume that \( p[x], \ q[x], \) and \( r[x] \) are recursive.

\((\forall x) \ \not p[x] \)

\((\forall x) \ \not \not [x] \)

\(*\not [p[x]] \)

\begin{align*}
\text{begin} & \quad \text{assume} \quad \not [p[x]] \\
& \quad s[x] = \text{cond(\not [p[x]])} \rightarrow F; \ \text{etc} \\
& \quad = F \\
& \quad \text{then} \quad *s[x] \quad \text{end} \\
\not [p[x]] & \rightarrow *s[x] \\
\text{begin} & \quad \text{assume} \quad \rightarrow [p[x]] \\
& \quad p[x] \\
& \quad *f[x] \)
\end{align*}
\begin{align*}
\text{begin assume } f(x) \\
\quad s(x) &= \text{cond}[\text{not}[p(x)] \rightarrow F; f(x) \rightarrow q(x); \text{ etc}] \\
\quad &= q(x) \\
\quad (\forall x) \ star q(x) \\
\text{then } *s(x) \text{ end}
\end{align*}

\begin{align*}
\text{f}(x) &\rightarrow *s(x) \\
\text{begin assume } \neg f(x) \\
\quad s(x) &= \text{cond}[\text{not}[p(x)] \rightarrow F; f(x) \rightarrow q(x); T \rightarrow r(x)] \\
\quad &= r(x) \\
\quad (\forall x) \ star r(x) \\
\text{then } *s(x) \text{ end}
\end{align*}

\begin{align*}
\neg f(x) &\rightarrow *s(x) \\
\text{then } *s(x) \text{ end}
\end{align*}

\begin{align*}
\neg \text{not}[p(x)] &\rightarrow *s(x) \\
(\forall x) \ star s(x)
\end{align*}

Next, we demonstrate $$(\forall x)(s(x) \leftrightarrow \text{cond}[f(x) \rightarrow g(x); T \rightarrow h(x)]$$

\begin{align*}
\text{begin assume } s(x) \\
\text{begin assume } \neg p(x) \\
\text{then } F \text{ end}
\end{align*}

\begin{align*}
p(x) \\
* f(x)
\end{align*}
\begin{verbatim}

begin assume \rightarrow f[x]
    
    s[x] = cond[\neg p[x]] \rightarrow F; f[x] \rightarrow q[x]; \rightarrow r[x]
          = r[x]

    \rightarrow r[x]

    \star h[x]

    cond[f[x] \rightarrow g[x]; \rightarrow h[x]] = h[x]

    then \star cond[f[x] \rightarrow g[x]; \rightarrow h[x]] end

\rightarrow f[x] \rightarrow \star cond[f[x] \rightarrow g[x]; \rightarrow h[x]]

begin assume f[x]

    s[x] = cond[\neg p[x]] \rightarrow F; f[x] \rightarrow q[x]; etc
          = q[x]

    \rightarrow q[x]

    \star g[x]

    cond[f[x] \rightarrow g[x]; \rightarrow h[x]] = g[x]

    then \star cond[f[x] \rightarrow g[x]; \rightarrow h[x]] end

f[x] \rightarrow \star cond[f[x] \rightarrow g[x]; \rightarrow h[x]]

    then \star cond[f[x] \rightarrow g[x]; \rightarrow h[x]] end

s[x] \rightarrow \star cond[f[x] \rightarrow g[x]; \rightarrow h[x]]

begin assume \star cond[f[x] \rightarrow g[x]; \rightarrow h[x]]

    \star f[x]

p[x]

begin assume f[x]
\end{verbatim}
\[ \text{cond} [f[x] \rightarrow g[x]; \ T \rightarrow h[x]] = g[x] \]

\[ \neg g[x] \]

\[ q[x] \]

\[ s[x] = \text{cond}[\text{not}[p[x]] \rightarrow F; \ f[x] \rightarrow q[x]; \ etc] \]

\[ = q[x] \]

\[ \text{then } s[x] \text{ end} \]

\[ f[x] \rightarrow s[x] \]

\[ \text{begin assume } \neg f[x] \]

\[ \text{cond} [f[x] \rightarrow g[x]; \ T \rightarrow h[x]] = h[x] \]

\[ \neg h[x] \]

\[ r[x] \]

\[ s[x] = \text{cond}[\text{not}[p[x]] \ F; f[x] \rightarrow g[x]; \ T \rightarrow r[x]] \]

\[ = r[x] \]

\[ \text{then } s[x] \text{ end} \]

\[ \neg f[x] \rightarrow s[x] \]

\[ \text{then } s[x] \text{ end} \]

\[ \neg \text{cond} [f[x] \rightarrow g[x]; \ T \rightarrow h[x]] \rightarrow s[x] \]

\[ (\forall x)(s[x] \leftrightarrow \neg \text{cond} [f[x] \rightarrow g[x]; \ T \rightarrow h[x]]) \]

**Theorem:** If \( f[x] = \text{cond}[p_1[x] \rightarrow e_1[x]; \ldots; p_n[x] \rightarrow e_n[x]] \) and all the \( p_i[x] \) and \( e_i[x] \) are potential recursive, then \( f[x] \) is potential recursive.
Proof: \( f[x] = \text{cond}[p_1[x] \leftrightarrow e_1[x]]; \)
\( T \leftrightarrow \text{cond}[p_2[x] \leftrightarrow e_2[x]]; \)
\( T \leftrightarrow \text{cond}[\ldots \]
\( T \leftrightarrow \text{cond}[p_n[x] \leftrightarrow e_n[x]] \ldots]] \)
and the result follows from repeated application of the previous three theorems.

Theorem: Suppose \( g[x;y] \) is defined primitive recursively in terms of \( s[x;y], k[x;y;a;b], m[y], \) and \( n[y]; \) i.e.,
\( g[x;y] = \text{cond}[\text{atom}[x] \leftrightarrow s[x;y]; \)
\( T \leftrightarrow k[x;y;g[\text{car}[x];m[y]];g[\text{cdr}[x];n[y]]]]. \)

Then, if the four functions \( s, k, m, \) and \( n \) are potential recursive with charfs \( sd, \) \( kd, \) \( md, \) and \( nd \) respectively, \( g \) is potential recursive and has charf
\( gd[x;y] = \text{cond}[\text{atom}[x] \leftrightarrow sd[x;y]; \)
\not[md[y]] \rightarrow F; \)
\not[gd[\text{car}[x]; m[y]]] \rightarrow F; \)
\not[nd[y]] \rightarrow F; \)
\not[gd[\text{cdr}[x]; n[y]]] \rightarrow F; \)
\( T \rightarrow kd[x;y;g[\text{car}[x];m[y]];g[\text{cdr}[x];n[y]]]]. \)

Theorem: Any function defined \( k \)-recursively in terms of potential recursive functions is a potential recursive function.
Chapter V

A MECHANICAL PROCEDURE FOR
FINDING CHARACTERISTIC FUNCTIONS

In this chapter we describe a computer program which was suggested by the results of the previous chapter. Generally speaking, this program produces (from the LISP definition of a function) a LISP definition of a characteristic function for the given function. But this flies in the face of several well-known results of recursive function theory: for there is no general procedure which can test a function for recursiveness, and there is no general procedure which can test whether a function has a recursive domain. (These problems are rated $\mathcal{P}_2$ and $\Sigma_3$, respectively, in the arithmetical hierarchy.) So, rather than resting with the broad, general, wrong description (above) of what this program does, we will give a very precise description.

The program is given in advance the recursive characteristic functions of all the basic LISP functions. (So, it "knows" that $\text{not}[\text{atom}[x]] \rightarrow *\text{car}[x]$.) In addition, the program is given names for all the characteristic functions it will require.

The program handles instances of substitution as in the following example: given $f[g[x];h[y]]$, the program calls itself to find characteristic functions for $g[x]$ and $h[y]$ (call these $q[x]$ and $r[y]$) and uses memorized information to determine the characteristic function for $f[a;b]$ (call this $p[a;b]$). Then the program gives as output the function

\[
\begin{align*}
\text{cond[not[r[y]]]} & \rightarrow F; \\
\text{not[q[x]]} & \rightarrow F; \\
T & \rightarrow p[g[x];h[y]].
\end{align*}
\]
Conditional expressions are handled as follows: given
\[ \text{cond}[p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n], \]
the program will call itself to determine the
characteristic functions of all the \( p_i \)'s (call these the \( q_i \)'s) and all the
\( e_i \)'s (call these the \( d_i \)'s). Then the program gives as output the function
\[
\text{cond}[\neg q_1 \rightarrow F; \\
p_1 \rightarrow d_1; \\
\neg q_2 \rightarrow F; \\
p_2 \rightarrow d_2; \\
\vdots \\
\neg q_n \rightarrow F; \\
p_n \rightarrow d_n; \\
T \rightarrow F].
\]

The program takes no notice of recursion. (Indeed, since the program
works with only one function at a time, recursion could be hidden from the
program. But this should be viewed as one of the program's definite shortcomings.)

The program "works" with potentially \( k \)-recursive functions and gives
mixed results for other functions. (We say \( f \) is potentially \( k \)-recursive
if there is a sequence of functions \( f_1, \ldots, f_n \) with \( f_n = f \) for which all of
the \( f_i \) are basic LISP functions, or defined in terms of some of the previous
functions by substitution, or defined (at most \( k \))-recursively in terms of
some of the previous functions, \( 1 \leq i \leq n \).) If the given function is potentially
\( k \)-recursive, then the program will produce a \( k \)-recursive characteristic
function.

But if the given function is not potentially \( k \)-recursive (for any \( k \)),
the program will produce a partial (although sometimes total) recursive
characteristic function. Suppose the given function is \( f[x] \), and the program
produces the function \( g[x] \). Then we have:
\[ *g[x] \land \neg g[x] \rightarrow \neg f[x] \]
\[ *g[x] \land \neg g[x] \rightarrow \neg f[x] \]
\[ \neg \neg g[x] \rightarrow \neg f[x] \]

At this point, a few examples should be of considerable interest. The first three examples are \texttt{ff[x]} (a recursive function), \texttt{morepar[x]} (a function defined for lists), and \texttt{append[x;y]} (as in Chapter III).

These listings are taken directly from a console attached to the Project MAC time-sharing system. The function \texttt{prettyprint} prints function definitions. The function \texttt{domain} finds the domain of the given function, and uses \texttt{prettyprint} to print the result.

\begin{verbatim}
prettyprint ((ff))

(FF
 (LAMBDA (X) (COND
   ((ATOM X) X)
   (T (FF (CAR X))))))

domain (ff)

(FFD
 (LAMBDA (X) (COND
   ((ATOM X) T)
   (T (FFD (CAR X))))))

prettyprint ((morepar))

(MOREPAR
 (LAMBDA (X) (COND
   ((NULL X) NIL)
   (T (CONS (CONS (CAR X) NIL) (MOREPAR (CDR X)))))))
\end{verbatim}
domain (morepar)

(MPDOM
  (LAMBDA (X) (COND
    ((NULL X) T)
    ((NOT (COND
      ((NOT (NOT (ATOM X))) F)
      (T (MPDOM (CDR X)))))) F)
    ((NOT (COND
      ((NOT (NOT (ATOM X))) F)
      (T T))) F)
    (T T)))))

prettyprint ((append))

(Append
  (LAMBDA (X Y) (COND
    ((NULL X) Y)
    (T (CONS (CAR X) (APPEND (CDR X) Y))))))

domain (append)

(AppendDom
  (LAMBDA (X Y) (COND
    ((NULL X) T)
    ((NOT (COND
      ((NOT (NOT (ATOM X))) F)
      (T (APPEND Dom (CDR X) Y)))))) F)
    ((NOT (NOT (ATOM X))) F)
    (T T)))))
The next function is double recursive, and has been shown to be non-primitive recursive. The domain of the characteristic function is also computed.

\texttt{pretyprint ((doub))}

\texttt{(DOUB}
\texttt{  (LAMBDA (N X) (COND}
\texttt{    ((ATOM N) (CONS X NIL)))
\texttt{    ((ATOM X) (DOUB (CAR N) (CONS X NIL)))
\texttt{    (T (DOUB (CAR N) (DOUB N (CAR X))))))}
\texttt{domain (doub)}

\texttt{(DOUBDOM}
\texttt{  (LAMBDA (N X) (COND}
\texttt{    ((ATOM N) T))
\texttt{    ((ATOM X) (COND}
\texttt{      ((NOT (NOT (ATOM N))) F)
\texttt{      (T (DOUBDOM (CAR N) (CONS X NIL)))))
\texttt{    ((NOT (COND}
\texttt{      ((NOT (NOT (ATOM X))) F)
\texttt{      (T (DOUBDOM N (CAR X)))))) F)
\texttt{    (T (DOUBDOM (CAR N) (DOUB N (CAR X)))))))}
\texttt{domain (doubdom)}

\texttt{(DOUBDOM2}
\texttt{  (LAMBDA (N X) (COND}
\texttt{    ((ATOM N) T))
\texttt{    ((ATOM X) (COND}
\texttt{      ((NOT (NOT (ATOM N))) T)
\texttt{      (T (DOUBDOM2 (CAR N) (CONS X NIL)))))
\texttt{    ((NOT (COND}
\texttt{      ((NOT (COND}
\texttt{        ((NOT (NOT (ATOM X))) T)
\texttt{        (T (DOUBDOM2 N (CAR X)))))) F)
\texttt{      (T T))))) F)
\texttt{    ((NOT (COND}
\texttt{      ((NOT (NOT (ATOM X))) F)
\texttt{      (T (DOUBDOM N (CAR X)))))) T)
\texttt{    (T (DOUBDOM2 (CAR N) (DOUB N (CAR X)))))))}
This function is the primitive recursion normal-form. We assume that
S, K, M and N have recursive characteristic functions SD, KD, MD and ND
respectively.

```
prettyprint ((prim))

(PRIM
 (LAMBDA (X Y) (COND
   ((ATOM X) (S X Y))
   (T (K X Y (PRIM
       (CAR X)
       (M Y)) (PRIM
       (CDR X)
       (N Y))))))))

domain (prim)

(PRIMDOM
 (LAMBDA (X Y) (COND
   ((ATOM X) (SD X Y))
   ((NOT (COND
       ((NOT (ND Y)) F)
       ((NOT (NOT (ATOM X))) F)
       (T (PRIMDOM
           (CDR X)
           (N Y)))))) F)
   ((NOT (COND
       ((NOT (MD Y)) F)
       ((NOT (NOT (ATOM X))) F)
       (T (PRIMDOM
           (CAR X)
           (M Y)))))) F)
   (T (KD X Y (PRIM
       (CAR X)
       (M Y)) (PRIM
       (CDR X)
       (N Y)))))))))
```
Finally, we describe the program. There are two major functions: domain and charf. The function charf uses the two subprograms chars and charcond to handle instances of substitution and conditional expressions, respectively. Both of these subprograms use csimp to simplify their output (when possible). The function fsimp (along with its subroutines formsub, lissub, consub, fsup, and fsup1) is used by csimp and charf to do lambda-conversion.

The function domain performs housekeeping chores for the program, while the function charf does the real work, and is the function which is called recursively (by its subprograms chars and charcond).

A function whose domain is known (or just "named") has on its property list a property called a DOMF. For all the basic LISP functions, the DOMF is the lambda-expression for the recursive characteristic function of the domain: e.g., get[car;DOMF] = (LAMBDA (X) (NOT (ATOM X))). For other functions, the DOMF is an atom which is the name of the characteristic function.

The function domain performs the following tasks: first, it makes sure that its argument is an atom with an EXPR and a DOMF on its property list. The DOMF must be an atom. If the DOMF has an EXPR on its property list, it is removed; and similarly for any DOMF on the property list of the DOMF. Then domain uses charf to find the characteristic function of its argument, places the result as an EXPR on the property list of the DOMF, and prettyprints the DOMF.

The function charf depends heavily on the presence of DOMFs and APVALs on the property lists of the things it is working with, since this is the only mechanism whereby it remembers which functions have known domains, or which unbound variables are defined.
For best results, this program should be used with functions which obey the following rules:

1) The function must be an EXPR.

2) The function may not use PROG or LABEL.

3) If any constants are used, they must have APVALs while domain is working.

4) It is advisable to bind every variable at every function call.

5) The function may not use functionals.

For details, the reader is referred to Appendix A. Two additional examples are included in Appendix B.

We feel that this program might form the nucleus of a new LISP debugging tool. It is certainly not suitable for such use as it stands now, but it should be possible to refine the program, improve the conditional-expression simplifier, add a recursiveness recognizer, etc.; and come up with something useful. (We do not expect this technique to be more fruitful than selective trace or break facilities, of course.)

Such improvements were not attempted as part of this research simply because this program is more interesting as a mathematical object.
Appendix A

The DOMAIN Program

DEFINE ((
  (DOMAIN (LAMBDA (FUNC) (PROG (U V W)
    (COND ((NOT (ATOM FUNC))
      (RETURN (QUOTE (ARGUMENT NOT AN ATOM))))))
    (SETQ U (GET (CDR FUNC) (QUOTE EXPR)))
    (COND ((NULL U) (RETURN (QUOTE (NO FUNCTION))))))
    (SETQ V (GET (CDR FUNC) (QUOTE DOMF))))
    (COND ((OR (NULL V) (NOT (ATOM V)))
      (RETURN (QUOTE (DOMF NOT NAMED))))))
    (REMPROP (CDR V) (QUOTE EXPR))
    (SETQ W V)
    (UNDEF
      (SETQ W (GET (CDR W) (QUOTE DOMF))))
    (COND ((OR (NULL W) (NOT (ATOM W)))
      (GO ALLU))
    (GO UNDEF))
    (REMPROP (CDR W) (QUOTE EXPR))
    (GO UNDEF)
    (ALLU
      (SETQ W (CHARF (CADR U) (CADDR U)))
      (ATTRIB (CDR V) (LIST (QUOTE EXPR) W))
      (PRETTYPRINT (LIST V))
      (RETURN BLANK) ))))

ATTRIB (CAR (DOMF (LAMBDA (X) (NOT (ATOM X))))))
ATTRIB (CDR (DOMF (LAMBDA (X) (NOT (ATOM X)))))
ATTRIB (CONS (DOMF (LAMBDA (X Y) T)))
ATTRIB (ATOM (DOMF (LAMBDA (X) T)))
ATTRIB (EQ (DOMF (LAMBDA (X Y) (AND (ATOM X) (ATOM Y)))))))
ATTRIB (NULL (DOMF (LAMBDA (X) T)))
ATTRIB (NOT (DOMF (LAMBDA (X) T)))
ATTRIB (AND (DOMF (LAMBDA (X Y) T)))
ATTRIB (OR (DOMF (LAMBDA (X Y) T)))
DEFINE ()

(CHARF (LAMBDA (VARLIST FORM) (COND
 ((ATOM FORM) (COND
  ((OR (MEMBER FORM VARLIST) (NULL FORM))
   (LIST (QUOTE LAMBDA) VARLIST (QUOTE T)))
  ((PROP (CDR FORM) (QUOTE APVAL)
     (QUOTE (LAMBDA () NIL)))
   (LIST (QUOTE LAMBDA) VARLIST (QUOTE T)))
  (T (LIST (QUOTE LAMBDA) VARLIST
     (LIST (QUOTE AMEMBER)
     (LIST (QUOTE QUOTE) FORM)
     (QUOTE (ALIST)) ) ) ) ) ))
((ATOM (CAR FORM)) (COND
 ((EQ (CAR FORM) (QUOTE QUOTE))
  (LIST (QUOTE LAMBDA) VARLIST (QUOTE T)))
 ((EQ (CAR FORM) (QUOTE COND))
  (CHARCOND VARLIST FORM))
 (T (CHARSUB VARLIST FORM)) ))
((EQ (CAAR FORM) (QUOTE LAMBDA))
 (CHARF VARLIST (FSIMP FORM)))
 (T (LIST (QUOTE LAMBDA) VARLIST
   (QUOTE (QUOTE (LURKING FUNCTIONAL))))) ) )

(CHARCOND (LAMBDA (VARLIST FORM) (PROG (U V W X))
 (SETQ X NIL)
 (SETQ U (CDR FORM))
 LOOP ((COND ((NULL U) (GO END)) )
  (SETQ V (CAR U))
  (SETQ W (CADDR (CHARF VARLIST (CAR V))))
  (COND ((EQ W (QUOTE T)) (GO SKIP))
    (SETQ X (CONS (LIST (LIST (QUOTE NOT) W) (QUOTE F)) X))
 SKIP (SETQ W (CADDR (CHARF VARLIST (CADR V))))
  (SETQ X (CONS (LIST (CAR V) W) X))
  (SETQ U (CDR U))
  (GO LOOP)
 END (SETQ X (REVERSE (CONS (QUOTE (T F)) X))))
 (RETURN (LIST (QUOTE LAMBDA) VARLIST (CSIMP X))))

(CHARSUB (LAMBDA (VARLIST FORM) (PROG (U V W))
 (SETQ U (GET (CDAR FORM) (QUOTE DOMF)))))
 (COND ((NULL U) (GO OOPS))
  (SETQ V (LIST (LIST (QUOTE T) (CONS U (CDR FORM))))))
 LOOP ((COND ((NULL U) (RETURN (LIST (QUOTE LAMBDA) VARLIST
   (CSIMP V) ))))
  (SETQ W (CADDR (CHARF VARLIST (CAR U))))
  (COND ((EQ W (QUOTE T)) (GO SKIP))
    (SETQ V (CONS (LIST (LIST (QUOTE NOT) W) (QUOTE F)) V))
 SKIP (SETQ U (CDR U))
  (GO LOOP)
 OOPS (PRINT (LIST (QUOTE (NO DOMF DEFINED FOR)) (CAR FORM)))
 (RETURN (LIST (QUOTE LAMBDA) VARLIST (QUOTE F)))) ) )
(CSIMP (LAMBDA (CL) (PROG (CLIST HYPLIST OLIST CLAUSE L) (SETQ CLIST (CDR (FSIMP (CONS (QUOTE COND) CL)))) (SETQ HYPLIST (QUOTE (F (NOT T) (NOT (NOT F))))) (SETQ OLIST NIL) LOOP (COND ((NULL CLIST) (GO END))) (SETQ CLAUSE (CAR CLIST)) (COND ((MEMBER (CAR CLAUSE) HYPLIST) (GO SKIPIT)) ((MEMBER (CAR CLAUSE) (QUOTE (T (NOT F) (NOT (NOT T))))) (GO OWISE)) ) ADDC (SETQ OLIST (CONS CLAUSE OLIST)) (SETQ HYPLIST (CONS (CAR CLAUSE) HYPLIST)) (SETQ HYPLIST (CONS (LIST (QUOTE NOT) (LIST (QUOTE NOT) (CAR CLAUSE))))) HYPLIST) ) SKIPIT (SETQ CLIST (CDR CLIST)) (GO LOOP) OWISE (COND ((EQ (CAADR CLAUSE) (QUOTE COND)) (GO NLIST))) (SETQ OLIST (CONS CLAUSE OLIST)) (GO END) NLIST (SETQ CLIST (CDADR CLAUSE)) (GO LOOP) END (SETQ L (LENGTH OLIST)) (COND ((EQUAL L 1) (COND ((EQ (CAAR OLIST) (QUOTE T)) (RETURN (CADAR OLIST))) (T (GO OOPS)))) ((EQUAL L 0) (GO OOPS)) (RETURN (CONS (QUOTE COND) (REVERSE OLIST)))) OOPS (PRINT (QUOTE (ERROR IN CSIMP))) (RETURN NIL) )
DEFINE ((
(FSIMP (LAMBDA (FORM) (FORMSUB (QUOTE X) (QUOTE X) FORM) )))

(FORMSUB (LAMBDA (E1 VAR E2) (COND
  ((ATOM E2) (COND
    ((EQ E2 VAR) E1)
    (T E2))
  ((ATOM (CAR E2)) (COND
    ((EQ (CAR E2) (QUOTE QUOTE)) E2)
    ((EQ (CAR E2) (QUOTE COND))
      (CONS (QUOTE COND) (CONS E1 VAR (CDR E2))))
    (T (CONS (FORMSUB E1 VAR (CAR E2))
       (LISSUB E1 VAR (CDR E2)))))
  (T (FSUB E1 VAR (CAR E2) (CDR E2))))))

(LISSUB (LAMBDA (E1 VAR E2) (COND
  ((NULL E2) NIL)
  (T (CONS (FORMSUB E1 VAR (CAR E2))
       (LISSUB E1 VAR (CDR E2)))))))

(CONSUB (LAMBDA (E1 VAR E2) (COND
  ((NULL E2) NIL)
  (T (CONS (LISSUB E1 VAR (CAR E2))
       (CONSUB E1 VAR (CDR E2)))))))

(FSUB (LAMBDA (E1 VAR FCN ARGS) (COND
  ((EQ (CAR FCN) (QUOTE LAMBDA)) (COND
    ((MEMBER VAR (CADR FCN))
     (FSUB1 (CDR FCN) (LISSUB E1 VAR ARGS)))
    (T (FORMSUB E1 VAR (FSUB1 (CDR FCN) ARGS)))))
  (T (CONS (FORMSUB E1 VAR FCN) (LISSUB E1 VAR ARGS))))))

(FSUB1 (LAMBDA (FCN ARGS) (COND
  ((NULL ARGS) (CADR FCN))
  (T (FSUB1 (LIST (CDAR FCN) (FORMSUB (CAR ARGS) (CAAR FCN)
        (CADR FCN))) (CDR ARGS)))))
))
Appendix B

Two Examples of Output from DOMAIN

prettyprint ((run))

(RUN
  (LAMBDA (X) (CONS X (RUN X))))

domain (run)

(RUNDOM
  (LAMBDA (X) (COND
    ((NOT (RUNDOM X)) F)
    (T T)))))

prettyprint ((doubrec))

(DOUBREC
  (LAMBDA (X Y Z) (COND
    ((OR
      (ATOM X)
      (ATOM Y)) (S X Y Z))
    (T (K X Y Z) (DOUBREC
      (CAR X)
      (M1 X Y Z (DOUBREC
        (CAR Y)
        (N1 Z)) (DOUBREC
        (CDR Y)
        (N2 Z)))
      (N3 Z)) (DOUBREC
        (CDR X)
      (M2 X Y Z (DOUBREC
        (CAR Y)
        (N4 Z)) (DOUBREC
        (CDR Y)
        (N5 Z)))
      (N6 Z)))))))
domain (doubrec)

(DOUBRECDOM
  (LAMBDA (X Y Z) (COND
    ((OR
      (ATOM X)
      (ATOM Y)) (SD X Y Z))
    ((NOT (COND
      ((NOT (ND6 Z)) F)
    ((NOT (COND
      ((NOT (ND5 Z)) F)
      ((NOT (NOT (ATOM Y))) F)
    (T (DOUBRECDOM
        X
        (CDR Y)
        (N5 Z)))))) F)
    ((NOT (COND
      ((NOT (ND4 Z)) F)
      ((NOT (NOT (ATOM Y))) F)
    (T (DOUBRECDOM
        X
        (CAR Y)
        (N4 Z)))))) F)
    (T (MD2 X Y Z (DOUBREC
        X
        (CAR Y)
        (N4 Z)) (DOUBREC
        X
        (CDR Y)
        (N5 Z)))))) F)
    ((NOT (NOT (ATOM X))) F)
  (T (DOUBRECDOM
      (CDR X)
      (M2 X Y Z (DOUBREC
        X
        (CAR Y)
        (N4 Z)) (DOUBREC
        X
        (CDR Y)
        (N5 Z)))
      (N6 Z)))))) F)
BIBLIOGRAPHY


