A Quick and Dirty Sketch of a Toy Logic

J Strother Moore

January 9, 2001

Abstract

For the purposes of this paper, a "logic" consists of a syntax, a set of axioms and some rules of inference. We define a simple Lisp-like logic, and then use it to prove a few theorems. The logic is not very precisely described, it is far too simple for our ultimate purposes, and it is not even sound! But presenting it gives us a vehicle for exploring the issues involved in defining a useful logic precisely.

1 Formal Logic

A logic is given by

- a syntax defining *terms* and *formulas*,
- an initial set of formulas called *axioms*, and
- some *rules of inference* allowing the derivation of new formulas from old.

A *proof* is a finite sequence of formulas called *theorems*. Each formula of the sequence is either an axiom or is derived from previous formulas in the sequence by a rule of inference. The last formula in the sequence is the theorem *proved* by the proof.

Our syntax will introduce terms representing variables, constants, and function applications. For example the term `(cons x y)` represents the application of the function symbol `cons` to the variable symbols `x` and `y`. Such a term is more traditionally written `cons`(x,y).

Terms will be combined into formulas using the equality operator, $=$, and the logical connectives $\neg$, $\vee$, $\rightarrow$ and $\wedge$.

What do these symbols mean? Intuitively, if you have an assignment of values to the variables and an assignment of functions to the function symbols, then every term evaluates to some object: variables evaluate to their assigned values and function applications evaluate to the result of applying the assigned function to the values of the argument terms. Thus, under an assignments of the variables and function symbols, every term has a *value*.

Every formula evaluates to either "true" or "false." To say that the value of $(x = y)$ is true is to say that the value of the term x is the same as that of the term y. To say that the value of $(\neg\phi)$ is true is to say that the value of ϕ is false. To say that the value of $(\phi_1 \vee \phi_2)$ is true is to say that either the value of ϕ_1 is true or the value of ϕ_2 is true. To say the value of $(\phi_1 \rightarrow \phi_2)$ is true is to say either that the value of ϕ_1 is false or the value of ϕ_2 is true. To say the value of $(\phi_1 \wedge \phi_2)$ is true is to say that the value of ϕ_1 is true and the value of ϕ_2 is true.

Some formulas are "valid" or "always true" in the sense that they are true no matter what values are assigned the variables in them. For example, $(\mathtt{x} = \mathtt{x})$ is always true. So is `(cdr (car x))` = `(cdr (car x))`. Clearly, if there are an infinite number of possible values for the variables, then you cannot determine validity in finite time by trying all the possibilities.

But now consider the following "happy coincidence." Suppose all the axioms in some logic are valid ("always true"). And suppose the rules of inference preserve validity. Such a logic is said to be "sound." Then the theorems — formulas derived from axioms and other theorems by rules of inference — are always true!

But the axioms and rules of inference are entirely syntactic entities: the axioms are just certain formulas and the rules are precisely described syntactic transformations of formulas. And proofs are finite objects. Thus, if you are given a formula in a sound logic and you want to establish that it is always true, you can try to derive it as a theorem, i.e., you can try to prove it. If you succeed you know the formula is always true! This is neat because a finite syntactic object, a proof, has replaced an infinite number of tests.[1]

This is the last we will talk about "semantics" or meaning of our terms and formulas. It suffices for our purposes merely to define the syntax of our terms and formulas, specify the set of axioms, give the rules of inference, and then learn how to find proofs.

2 Syntax

Formulas are built from terms. Terms are built from symbols. We first describe the symbols, then we explain how to build up terms, and finally we explain how to build formulas.

We use without definition the notion of a "symbol." Examples of symbols are `x`, `y23`, `nil` and `count`.

The *constant symbols* are the symbols `t` and `nil`. All other symbols are

[1] Suppose you fail to find a proof. What do you know about the truth of the formula? In general, you've learned nothing. It could sometimes be false. Or it could in fact be a theorem, and thus always true, but the proof has just eluded you. Or, perhaps, the formula is always true but cannot be derived from the given axioms and rules because they are "weak" or "incomplete."

function symbol	*arity*	*comment*
`equal`	2	equality function
`car`	1	first component of a pair
`cdr`	1	second component of a pair
`cons`	2	constructs ordered pairs
`consp`	1	recognizes ordered pairs
`if`	3	if-then-else

Table 1: The Primitive Function Symbols

variable symbols. *Function symbols* are symbols that have an associated *arity* indicating how many arguments the corresponding function takes. For the present purposes, the function symbols and their arities are given in Table 1. But we will add new ones as we go.

A *term* is either a variable symbol, a constant symbol, or a sequence consisting of a function symbol of arity n followed by n terms.

Non-variable, non-constant terms denote the *application* of the indicated function to n *argument* terms. Function applications are written by writing down the applied function symbol and the argument terms, separating them by white space and enclosing the whole sequence in parentheses. Thus, `(cons x (cdr y))` is a term denoting the application of the function `cons` to the two arguments `x` and `(cdr y)`. The first argument is a variable. The second denotes the application of the function `cdr` to the variable `y`. In high school we learned to write this term cons(x, cdr(y)).

It is convenient to adopt the convention that `(list` x_1 x_2 ... x_n`)` abbreviates the term `(cons` x_1 `(cons` x_2 ... `(cons` x_n `nil)...))`. Thus, for example, `(list a b c)` abbreviates `(cons a (cons b (cons c nil)))`.

Now we move on to formulas.

An *atomic formula* is any sequence of the form $(t_1 = t_2)$, where t_1 and t_2 are terms. A *formula* is either an atomic formula, or else of the form $(\neg\phi)$, where ϕ is a formula, or else of the form $(\phi_1 \vee \phi_2)$, where ϕ_1 and ϕ_2 are both formulas. Parentheses are often omitted from formulas (but *never* from terms) when no ambiguity arises. For example, we might write the atomic formula (`(cons x y)` = `(cons u v)`) more simply as `(cons x y)` = `(cons u v)`.

A *substitution* is a mapping from variable symbols to terms. Technically, a substitution is a set of ordered pairs. But we write them in a special way to make clear what variables are being replaced by what. Here is the substitution that replaces the variable symbol `x` by `(f x y)` and the variable symbol `y` by the variable symbol `a`: { `x` ◁ `(f x y)` ; `y` ◁ `a` }.

To *apply* a substitution to a term (or formula), one uniformly replaces all occurrences of the mapped variable symbols by the corresponding terms. The term (or formula) obtained by applying a substitution σ to a term (or formula) ϕ is denoted ϕ/σ. A term (or formula) is an *instance* of another if and only if

the former can be obtained from the latter by applying a substitution.

For example, let σ be { `x` $\lhd$ `(f x y)` ; `y` $\lhd$ `a` }. Then `(cons x (cdr (h a y)))`$/\sigma$ is `(cons (f x y) (cdr (h a a)))` The term `(cons (f x y) (cdr (h a a)))` is an instance of `(cons x (cdr (h a y)))`.

Function and variable symbols are generally written in `typewriter` font. Italized Roman letters are generally used as meta-variables standing for terms and function symbols. Thus, if we say "the term `(f x)`" we mean the explicit term representing the function symbol `f` applied to the variable symbol `x`. But if we say "a term of the form $(f\ x)$" we mean any term in which a unary function symbol, here denoted by f, is applied to any argument term x (which may or may not be a variable symbol). We generally use Greek letters as meta-variables standing for formulas and for substitutions.

When $(t_1 \neq t_2)$ is used as a formula it is an abbreviation for the formula $(\neg(t_1 = t_2))$. When $(\phi_1 \rightarrow \phi_2)$ is used as a formula, it is an abbreviation for $(\neg\phi_1 \vee \phi_2)$. When $(\phi_1 \wedge \phi_2)$ is used as a formula, it is an abbreviation for the formula $\neg(\neg\phi_1 \vee \neg\phi_2)$. When $(\phi_1 \leftrightarrow \phi_2)$ is used as a formula, it is an abbreviation for the formula abbreviated by $(\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1)$.

When we use a term t where a formula is expected, we mean to use the formula $t \neq$ `nil` instead. We call this our *term-as-formula convention*.

For example, it is technically incorrect to refer to (`p` $\vee$ `(car x)`) as a "formula" — `p` and `(car x)` are terms. Such usage is understood to refer to the formula (`p`$\neq$`nil` $\vee$ `(car x)`$\neq$`nil`).

3 Axioms

We now identify all the formulas that are "axioms." We divide the axioms into two groups. The first group, called the "propositional equality" axioms, provide us with propositional calculus and equality. The second group, called the "Lisp axioms," introduce the properties of various function symbols.

3.1 Propositional Equality

Any classical formalization of propositional calculus and equality will suit our purposes. So that this "toy logic" is self-contained we have included one such formalization, essentially that of Shoenfield [1]. Shoenfield formalizes propositional calculus with one axiom schema and four rules of inference. He introduces equality with three axiom schemas.

Axiom Schema (the *Propositional Axiom*).
$(\neg\phi \ \vee \ \phi)$

By this we mean every formula of the above form is an axiom.

Axiom (*Reflexivity*).
$(x \ = \ x)$

Axiom Schema (*Equality Axioms for Functions*).
For every function symbol f of arity n we add:

((x1 = y1) →
...
((xn = yn) →

(*f* x1 ... xn) = (*f* y1 ... yn))...)

An example of the above Equality Axiom is that for `cons`,

((x1 = y1) → ((x2 = y2) → (cons x1 x2) = (cons y1 y2))).

Axiom. (*Equality Axiom for* =)
((x1=y1) → ((x2=y2) → ((x1=x2) → (y1=y2)))).

Recall that the rules of inference of our logic (not yet presented) will permit the derivation of theorems from these axioms. The derivation of theorems is the formal analogue of "reasoning." It may be hard to believe that the axioms presented above can be combined using a few rules to do all propositional reasoning and all reasoning about the equality of terms composed of uninterpreted function symbols. And yet that is the case! If this isn't already known to you, then just take it on faith for the moment. We will demonstrate how to do propositional and equality reasoning with our axioms and rules after we have finished introducing the other axioms and the rules.

3.2 Lisp Axioms

We have the following axioms describing the properties of particular constants and function symbols.

Axiom 1.
t ≠ nil

Axiom 2.
x = y → (equal x y) = t

Axiom 3.
x ≠ y → (equal x y) = nil

Axiom 4.
x = nil → (if x y z) = z

Axiom 5.
x ≠ nil → (if x y z) = y.

Axiom 6.
(consp (cons x y)) = t

Axiom 7.

`(consp nil) = nil`

Axiom 8.

`(consp t) = nil`

Axiom 9.

`(car (cons x y)) = x`

Axiom 10.

`(cdr (cons x y)) = y`

Axiom 11.

`(consp x) = t` $\rightarrow$ `(cons (car x) (cdr x)) = x`

4 Rules of Inference

The rules of inference are extraordinarily simple. The first five are taken from [1] and support propositional calculus and equality reasoning.

Rules of Inference.

- *Expansion*: derive $(\phi_1 \vee \phi_2)$ from ϕ_2;
- *Contraction*: derive ϕ from $(\phi \vee \phi)$;
- *Associativity*: derive $((\phi_1 \vee \phi_2) \vee \phi_3)$ from $(\phi_1 \vee (\phi_2 \vee \phi_3))$; and
- *Cut*: derive $(\phi_2 \vee \phi_3)$ from $(\phi_1 \vee \phi_2)$ and $(\neg\phi_1 \vee \phi_3)$.

Rule of Inference. *Instantiation*:
Derive ϕ/σ from ϕ.

Rule of Inference. *Definition*:
Derive `(`$f\ v_1 \ldots v_n$`)` $=$ *body*,

provided that

- f is a new function symbol of arity n (by *new* we mean f is not mentioned in any axiom or theorem heretofore introduced);
- the v_i are variable symbols; and
- *body* is a term.

The rule of inference above permits us to define a new function symbol by adding a new axiom.

Rule of Inference. *Induction*:
Derive ϕ from

- *Base Case*. $(\texttt{(consp } x\texttt{)} = \texttt{nil}) \rightarrow \phi$
- *Induction Step*. $(\texttt{(consp } x\texttt{)} = \texttt{t}) \wedge \phi/\sigma) \rightarrow \phi$,

where σ is any substitution that replaces x by `(cdr` x`)`. The substitution may replace other variables arbitrarily.

Probably you are most familiar with induction in the setting of the natural numbers. To prove a conjecture for all natural numbers n, the base case requires that we prove it when $n = 0$. The induction step provides the conjecture for n as a hypothesis and requires us to prove it for $n + 1$. But we do not have the natural numbers in this toy logic. We have lists built by `cons` and "recognized" by `consp`. Here, our base case is for non-conses, i.e., objects x not built by `cons`. Our induction step has us assume x is built by `cons` and gives us the hypothesis that the conjecture holds for the `cdr` of x. We must prove that it holds for x. This formulation of induction is too weak to meet our needs in general, but will get us off the ground with inductive proofs.

5 Getting Started with Formal Proofs

5.1 Propositional Calculus

We will start with some simple propositional calculus.[2] First, we will prove the theorem `(P` $\vee$ `Q)` $\rightarrow$ `(Q` $\vee$ `P)`. However, the formal proof of even so simple a theorem is very long if we limit ourselves to the rules of inference given, namely Expansion, Contraction, Associativity, and Cut. We therefore start by deriving several new rules of inference.

Derived Rule of Inference. *(Commutativity of Or)*
Derive $(\beta \vee \alpha)$ from $(\alpha \vee \beta)$.

Below we derive $(\beta \vee \alpha)$ from $(\alpha \vee \beta)$. The proof is presented as a sequence of formulas, each of which is either an axiom or is given (in the case of a derived rule of inference) or is derived by inference rules from previous formulas. We number and justify each formula in the presentation below. Unless otherwise noted, each rule of inference takes as its single premise the formula immediately above.

[2]Most of the propositional calculus proofs were contributed by N. Shankar.

Proof.

1.	$(\alpha \vee \beta)$	Given
2.	$(\neg(\alpha) \vee \alpha)$	Propositional Axiom
3.	$(\beta \vee \alpha)$	Cut, lines 1 and 2
	Q.E.D.	

What exactly is going on here? We have shown that from $(\alpha \vee \beta)$ we can derive $(\beta \vee \alpha)$ by applying the primitive rules, the Propositional Axiom and Cut. Thus, in the future, if we have derived a formula $(\phi_1 \vee \phi_2)$ we can, in the next line, derive $(\phi_2 \vee \phi_1)$ and attribute the derivation to the new rule Commutativity of Or, in the knowledge that we could convert the resulting "proof" into a proof by substituting for that single line the steps above.

We now derive several other useful rules, using Commutativity of Or:

Derived Rule of Inference. *(Or Insertion 1)*
Derive $\alpha \vee (\gamma \vee \beta)$ from $(\alpha \vee \beta)$.

Proof.

1.	$(\alpha \vee \beta)$	Given
2.	$(\beta \vee \alpha)$	Commutativity of Or
3.	$(\gamma \vee (\beta \vee \alpha))$	Expansion
4.	$((\gamma \vee \beta) \vee \alpha)$	Associativity
5.	$\alpha \vee (\gamma \vee \beta)$	Commutativity of Or
	Q.E.D.	

Derived Rule of Inference. *(Or Insertion 2)*
Derive $\alpha \vee (\beta \vee \gamma)$ from $(\alpha \vee \beta)$.

Proof.

1.	$(\alpha \vee \beta)$	Given
2.	$(\beta \vee \alpha)$	Commutativity of Or
3.	$(\beta \vee (\gamma \vee \alpha))$	Or Insertion 1
4.	$((\beta \vee \gamma) \vee \alpha)$	Associativity
5.	$\alpha \vee (\beta \vee \gamma)$	Commutativity of Or
	Q.E.D.	

Derived Rule of Inference. *(Or Implication)*
Derive $(\alpha \vee \beta) \rightarrow \gamma$ from $(\alpha \rightarrow \gamma)$ and $(\beta \rightarrow \gamma)$.

Proof.

1.	$\beta \to \gamma$	Given
2.	$\neg\beta \vee \gamma$	Abbreviation
3.	$\neg(\alpha \vee \beta) \vee (\alpha \vee \beta)$	Propositional Axiom
4.	$(\neg(\alpha \vee \beta) \vee \alpha) \vee \beta$	Associativity
5.	$\beta \vee (\neg(\alpha \vee \beta) \vee \alpha)$	Commutativity of Or
6.	$(\neg(\alpha \vee \beta) \vee \alpha) \vee \gamma$	Cut, lines 5 and 2
7.	$\gamma \vee (\neg(\alpha \vee \beta) \vee \alpha)$	Commutativity of Or
8.	$(\gamma \vee \neg(\alpha \vee \beta)) \vee \alpha$	Associativity
9.	$\alpha \vee (\gamma \vee \neg(\alpha \vee \beta))$	Commutativity of Or
10.	$\alpha \to \gamma$	Given
11.	$\neg\alpha \vee \gamma$	Abbreviation
12.	$\neg\alpha \vee (\gamma \vee \neg(\alpha \vee \beta))$	Or Insertion 2
13.	$(\gamma \vee \neg(\alpha \vee \beta)) \vee (\gamma \vee \neg(\alpha \vee \beta))$	Cut, lines 9 and 12
14.	$(\gamma \vee \neg(\alpha \vee \beta))$	Contraction
15.	$\neg(\alpha \vee \beta) \vee \gamma$	Commutativity of Or
16.	$(\alpha \vee \beta) \to \gamma$	Abbreviation
	Q.E.D.	

We are finally in a position to prove our first theorem!

Theorem. $(\mathtt{P} \vee \mathtt{Q}) \to (\mathtt{Q} \vee \mathtt{P})$.

Before we prove this, note that technically this is not a formula because terms `P` and `Q` are used as formulas. We understand the "formula" above to mean $(\mathtt{P} \neq \mathtt{nil} \vee \mathtt{Q} \neq \mathtt{nil}) \to (\mathtt{Q} \neq \mathtt{nil} \vee \mathtt{P} \neq \mathtt{nil})$. But we use our term-as-formula convention freely in the proof below.

Proof.

1.	$\neg\mathtt{P} \vee \mathtt{P}$	Propositional Axiom
2.	$\neg\mathtt{P} \vee (\mathtt{Q} \vee \mathtt{P})$	Or Insertion 1
3.	$\mathtt{P} \to (\mathtt{Q} \vee \mathtt{P})$	Abbreviation
4.	$\neg\mathtt{Q} \vee \mathtt{Q}$	Propositional Axiom
5.	$\neg\mathtt{Q} \vee (\mathtt{Q} \vee \mathtt{P})$	Or Insertion 2
6.	$\mathtt{Q} \to (\mathtt{Q} \vee \mathtt{P})$	Abbreviation
7.	$(\mathtt{P} \vee \mathtt{Q}) \to (\mathtt{Q} \vee \mathtt{P})$	Or-Implication, lines 3 and 6
	Q.E.D.	

The following classic derived rule is very useful.

Derived Rule of Inference. *(Modus Ponens)*
Derive β from α and $\alpha \to \beta$.

Proof.

1.	α	Given
2.	$\beta \vee \alpha$	Expansion
3.	$\alpha \vee \beta$	Commutativity of Or
4.	$\alpha \rightarrow \beta$	Given
5.	$\neg\alpha \vee \beta$	Abbreviation
6.	$\beta \vee \beta$	Cut, lines 3 and 5
7.	β	Contraction

Q.E.D.

There are, of course, many other derived rules of inference about propositional calculus. Among them are versions of the tautology theorem ("every valid propositional formula has a proof"), the deduction law ("assume as given the hypotheses of an implication and prove the conclusion"), and case analysis ("prove the theorem under an exhaustive set of cases").

Having shown how to "get off the ground" using Shoenfield's system, we will henceforth use such propositional rules and results freely. You are free to use such results in your proofs for this class. Readers unfamiliar with propositional calculus are urged to consult a logic textbook, e.g., [1].

5.2 Equality

We next prove a simple theorem about the equality predicate. We are not so much interested in the theorem as in the proof, because it suggests the proof of the commonly used derived rule of "substitution of equals for equals."

Theorem.
`A` $=$ `B` $\rightarrow$ `(CAR (CDR A))` $=$ `(CAR (CDR B))`

Proof.

1.	`A` $=$ `B` $\rightarrow$ `(CDR A)` $=$ `(CDR B)`	Instantiation of Equality Axiom for `CDR`
2.	$\neg$ (`A` $=$ `B`) $\vee$ `(CDR A)` $=$ `(CDR B)`	Abbreviation
3.	`(CDR A)` $=$ `(CDR B)` $\vee \neg$ (`A` $=$ `B`)	Commutativity of Or
4.	`(CDR A)` $=$ `(CDR B)` $\rightarrow$ `(CAR (CDR A))` $=$ `(CAR (CDR B))`	Instantiation of Equality Axiom for `CAR`
5.	$\neg$ (`(CDR A)` $=$ `(CDR B)`) $\vee$ `(CAR (CDR A))` $=$ `(CAR (CDR B))`	Abbreviation
6.	$\neg$ (`A` $=$ `B`) $\vee$ `(CAR (CDR A))` $=$ `(CAR (CDR B))`	Cut, lines 3 and 5
7.	`A` $=$ `B` $\rightarrow$ `(CAR (CDR A))` $=$ `(CAR (CDR B))`	Abbreviation

Q.E.D.

By induction on the structure of terms we can derive the following powerful and widely used rule of inference:

Derived Rule of Inference. *(Substitution of Equals for Equals)*
If $a = b$ has been proved and formula β is obtained from formula α by replacing some occurrences of a in α by b, then β is provable iff α is provable.

By combining this rule with the deduction law we can substitute b for selected occurrences of a in ϕ when trying to prove a formula of the form $(a = b) \rightarrow \phi$.

Henceforth, we freely use equality reasoning in our proofs.

6 Defining the Propositional Functions

To use formal logic to model some system it is usually necessary to extend the logic by defining new concepts. This is done by adding new axioms with the Definition rule of inference, sometimes called the *Definitional Principle*.

Below are definitions, defining five new function symbols. Implicitly, we should extend the "arity table" (Table 1) appropriately.

Defining Axiom
`(not p) = (if p nil t)`

Defining Axiom
`(and p q) = (if p (if q t nil) nil)`

Defining Axiom
`(or p q) = (if p t (if q t nil))`

Defining Axiom
`(implies p q) = (if p (if q t nil) t)`

Defining Axiom
`(iff p q) = (and (implies p q) (implies q p))`

Now let's prove a few key theorems about these new symbols.

Theorem. *(*`if` *cases)*
`(if x y z)` = `u` $\leftrightarrow$ (`x`$\neq$`nil` $\rightarrow$ `y`=`u`) $\wedge$ (`x`=`nil` $\rightarrow$ `z`=`u`)

Proof.
Name the formula above *1. The proof of *1 is by case analysis on `x`=`nil`.

Case 1. `x`=`nil`.
In this case, by **Axiom 4**, `(if x y z)` = `z`. Thus, the left-hand side of *1 is `z`=`u`. The right-hand side becomes

(`nil`$\neq$`nil` $\rightarrow$ `y`=`u`) $\wedge$ (`nil`=`nil` $\rightarrow$ `z`=`u`)

By propositional calculus and equality, this is equivalent to `z`=`u`.

Case 2. `x` $\neq$ `nil`.
By **Axiom 5**, the left-hand side of *1 is equivalent to `y=u`. By propositional calculus and equality, the right-hand side also equivalent to `y=u`.
Q.E.D.

Theorem. *(*`implies` *is implication)*

`(implies p q)`$\neq$ `nil` $\leftrightarrow$ `(p`$\neq$`nil` $\rightarrow$ `q`$\neq$`nil)`

Proof.
We will repeatedly replace the left-hand side of the above equivalence by equivalent formulas until the result is identical to the right-hand side. By the definition of `implies` (**Axiom 15**), the left-hand side above is equivalent to

`(if p (if q t nil) t)`$\neq$`nil`

which, by `if` cases, is equivalent to

`(p`$\neq$`nil` $\rightarrow$ `(if q t nil)`$\neq$`nil)` $\wedge$ `(p=nil` $\rightarrow$ `t`$\neq$`nil).`

By **Axiom 1** and propositional calculus, the above is equivalent to

`p`$\neq$`nil` $\rightarrow$ `(if q t nil)`$\neq$`nil.`

Applying `if` cases again produces

`p`$\neq$`nil` $\rightarrow$ `((q`$\neq$`nil` $\rightarrow$ `t`$\neq$`nil)` $\wedge$ `(q=nil` $\rightarrow$ `nil`$\neq$`nil))`

which, by **Axiom 1** and propositional calculus with equality, is equivalent to

`(p`$\neq$`nil` $\rightarrow$ `q`$\neq$`nil).`

Q.E.D.

This style of proof is so common that we frequently abbreviate it as follows.

`(if p (if q t nil) t)`$\neq$`nil`
$\leftrightarrow$
`(p`$\neq$`nil` $\rightarrow$ `(if q t nil)`$\neq$`nil)` $\wedge$ `(p=nil` $\rightarrow$ `t`$\neq$`nil).`
$\leftrightarrow$
`p`$\neq$`nil` $\rightarrow$ `(if q t nil)`$\neq$`nil.`
$\leftrightarrow$
`p`$\neq$`nil` $\rightarrow$ `((q`$\neq$`nil` $\rightarrow$ `t`$\neq$`nil)` $\wedge$ `(q=nil` $\rightarrow$ `nil`$\neq$`nil))`
$\leftrightarrow$
`(p`$\neq$`nil` $\rightarrow$ `q`$\neq$`nil)`

We omit the explanatory justifications when they are sufficiently "obvious."

Derived Rule of Inference. *(Term based Modus Ponens)*
The term q may be derived as a theorem if the term p is a theorem and the term `(implies p q)` is a theorem. The proof is trivial, given `implies` is implies.

Theorem. *(*`not` *case)*

`(not p)` $\neq$`nil` $\leftrightarrow$ `p=nil`

Proof. By the definition of `not` (**Axiom 12**), the left-hand side is equivalent to `(if p nil t)`$\neq$`nil` which, by `if` cases, is equivalent to

`(p` $\neq$`nil` $\rightarrow$ `nil` $\neq$`nil)` $\wedge$ `(p=nil` $\rightarrow$ `t` $\neq$`nil)`.

But by **Axiom 1** and propositional calculus with equality, this is equivalent to `p=nil`. **Q.E.D.**

Theorem. *(*`equal` *is equality)*

`(equal x y)` $\neq$`nil` $\leftrightarrow$ `x=y`

Proof. That the left-hand side implies the right is merely the contrapositive of **Axiom 3**. That the right-hand side implies the left follows from **Axioms 1** and **2** by equality reasoning. **Q.E.D.**

Analogous theorems hold for `not`, `and`, `or` and `iff`.

These theorems, together with our term-as-formula convention, justify our naming these new functions the "propositional functions." For example, by the convention, when the term

```
(implies (and p (equal x y)) r)
```

is used as a formula, it is equivalent to

`(implies (and p (equal x y)) r)` $\neq$ `nil`

which is equivalent, by the theorems just mentioned, to

`((p` $\neq$ `nil)` $\wedge$ `(x=y))` $\rightarrow$ `r` $\neq$`nil`

which, by our convention, may be abbreviated

`(p` $\wedge$ `(x=y))` $\rightarrow$ `r`

Henceforth, we will reason about the propositional functions `equal`, `if`, `not`, `and`, `or`, `implies` and `iff` with the same casual ease we use with propositional calculus and equality.

7 Computing with Axioms and Substitutions

The proof of the following theorem shows how we can "compute" within the logic. Computation is just the systematic use of function definition axioms and the replacement of equals by equals.

Theorem.

`(and t (or nil t)) = t.`

Proof.

```
(and t (or nil t))
=
(and t (if nil t (if t t nil)))
=
(and t (if t t nil))
=
(and t t)
=
(if t (if t t nil) nil)
=
(if t t nil)
=
t
```

Q.E.D.

The basic idea is to focus repeatedly on the innermost function application whose arguments are all constants, "expand" it by replacing it by the instantiated "body" of its definition, and then "reduce" the more primitive expression using the basic axioms and the definitions of its "subroutines."

We often compress proofs as above into "one liners." We might say "`(and t (or nil t))` = `t`, by computation (or evaluation)."

8 A Simple Recursive Definition

Consider the new axiom:

Defining Axiom
`(true x) = (if (consp x) (true (cdr x)) t)`

This function always returns `t`. But let's start our investigation of the function by proving a simple lemma about it.

Theorem.

`(true (cons x y)) = (true y).`

Proof.

```
(true (cons x y))
=                          {by the definition of true }
(if (consp (cons x y))
    (true (cdr (cons x y)))
    t)
=                          {by  Axiom 6 and equality}
```

```
(if t
    (true (cdr (cons x y)))
    t)
```
= {by **Axiom 5** and equality}
```
(true (cdr (cons x y)))
```
= {by **Axiom 10** and equality}
```
(true y)
```
Q.E.D.

This style of proof essentially the same as "computation" but is sometimes called *symbolic computation* or *symbolic evaluation* to acknowledge the presence of variable symbols in the "data."

Similarly, we can prove

Theorem.
`(true (list x y z)) = t`

Proof.
```
(true (list x y z))
```
= {by the `list` abbreviation}
```
(true (cons x (cons y (cons z nil))))
```
= {by symbolic computation}
```
(true (cons y (cons z nil)))
```
= {by symbolic computation}
```
(true (cons z nil))
```
= {by symbolic computation}
```
(true nil)
```
= {by computation}
`t`

Q.E.D.

We would generally just compress this to one line and say "`(true (list x y z)) = t` by symbolic computation."

Obviously, we can show by symbolic computation that `true` returns `t` on any argument of the form `(list` x_1 ... x_n`)`. But how do we show that it returns `t` on all x? We might start the proof as follows:

Theorem.
`(true x) = t`

Proof.
Case 1. `(consp x)` = `nil`. `(true x)` = `t` by computation.
Case 2. `(consp x)` $\neq$ `nil`. `(true x)` = `(true (cdr x))`, by symbolic computation.

Case 2.1. `(consp (cdr x))` = `nil`. `(true (cdr x))` = `t` by computation.
Case 2.2. `(consp (cdr x))` $\neq$ `nil`. `(true (cdr x))` = `(true (cdr (cdr x)))` by symbolic computation.
Case 2.2.1. `(consp (cdr (cdr x)))` = `nil`. ...

But this "proof" will clearly require an infinite number of case splits and so won't succeed. Induction is required to prove this theorem and the failed proof attempt clearly suggests why.

Induction is like case analysis but provides, in the "interesting" case (the second case above), an additional hypothesis about `(cdr x)`. And in the second case above, we reduce our proof obligation from `(true x)` to `(true (cdr x))` because `true` is defined recursively.

9 A Simple Inductive Proof

Theorem.
`(true x)` = `t`

Proof.
The proof is by induction on `x`, i.e., substitution σ is { `x` $\lhd$ `(cdr x)` }.
Base Case.
`((consp x)` = `nil)` $\rightarrow$ `(true x)` = `t`.
This is obvious by computation. Here is the "obvious" proof. Using the deduction law, we can assume the hypothesis and prove the conclusion under that assumption. That is,

`(consp x)` = `nil`	{Given as induction case analysis}

Then we work on the conclusion, by reducing the left-hand side to the right.

`(true x)`	{left-hand side}
=	{by symbolic computation}
`t`.	{right-hand side}

Induction Step.
`(((consp x)` = `t)` $\wedge$ `((truep (cdr x))` = `t))` $\rightarrow$ `(truep x)` = `t`
This too is obvious by symbolic computation. But here is the "obvious" proof.

`(consp x)` = `t`	{Given as induction case analysis}
`(true (cdr x))` = `t`	{Given as induction hypothesis}

We now work on the concludion of the induction step:

`(true x)`	{left-hand side}
=	{by symbolic computation}
`(true (cdr x))`	
=	{by induction hypothesis}
`t`.	{right-hand side}

Q.E.D.

Eventually, proofs such as this will be described in one line such as "the proof is by induction on `x`." But for the moment proofs should be written at the level of detail shown above.

10 Problems

Not all of the "theorems" below are theorems! If you are asked to prove a non-theorem, you should, at least, exhibit a counterexample, i.e., show values for the variables that make the value of the formula `nil`. Often these non-theorems will suggest theorems to you. That is, there is a germ of an interesting idea in the non-theorem and it just needs to be said more precisely to be valid. When you see the "interesting idea" in a non-theorem, re-state it as a theorem. Then prove it!

You may find that the logic is too weak to allow you to prove certain things. If so, propose improvements.

You may find that the logic is too strong – i.e., allows you to prove things you should not be able to prove. If so, propose refinements.

Problem 1.1. Define the function `app` to concatenate two arbitrarily long lists, i.e., `(app (list` x_1 ... x_n`) (list` y_1 ... y_k`))` is equal to `(list` x_1 ... x_n y_1 ... y_k`)`.

Problem 1.2. Prove `(app nil y)` = `y`.

Problem 1.3. Prove `(app (list a b c) (list x y))` = `(list a b c x y)`.

Problem 1.4. Prove that `app` is associative.

Problem 1.5. State and prove that "`nil` is a right identity for `app`."

Problem 1.6. Here is a definition.

Defining Axiom

```
(rev x)
  =
(if (consp x)
    (app (rev (cdr x)) (list (car x)))
    nil)
```

What is the value of `(rev (list a b c d))`?

Problem 1.7 Prove `(rev (rev x))` = `x`.

Problem 1.8. Here is a definition.

Defining Axiom

```
(rev1 x a)
  =
(if (consp x)
    (rev1 (cdr x) (cons (car x) a))
    a)
```

What is the value of `(rev1 (list a b c d) nil)`?

Problem 1.9. Prove `(rev1 x nil)` = `(rev x)`.

Problem 1.10. Define `(mem e x)` to return `t` or `nil` according to whether `e` is an element of the list `x`.

Problem 1.11. Prove `(mem e (app a b))` = `(or (mem e a) (mem e b))`.

Problem 1.12. Define `(subsetp x y)` to return `t` or `nil` according to whether every element of the list `x` is an element of the list `y`.

Problem 1.13. Prove that `subsetp` is reflexive.

Problem 1.14. Prove that `subsetp` is transitive.

Problem 1.15. Prove `(subsetp a (app a b))`.

Problem 1.16. Prove `(subsetp b (app a b))`.

Problem 1.17. Here is a definition.

Defining Axiom

```
(f x)
  =
(if (consp x)
    (app (f (car x)) (f (cdr x)))
    (list x))
```

Explain what this function computes. Think of `(cons` a b`)` as a binary tree whose left branch is the subtree a and whose right branch is the subtree b.

Problem 1.18. What does this function do?

Defining Axiom

```
(mcf x a)
  =
(if (consp x)
    (mcf (car x)
         (mcf (cdr x) a))
    (cons x a))
```

Problem 1.19. Prove `(mcf x nil)` = `(f x)`.

Problem 1.20. Define `(occ e x)` to return `t` or `nil` according to whether `e` occurs as a tip of the binary tree `x`.

Problem 1.21. Prove `(mem e (f x))` = `(occ e x)`.

Problem 1.22. The *length* of x is 0 if x is not a cons and is one more than the length of `(cdr` x`)`, otherwise. Define `(<` `x` `y)` to return `t` or `nil` according to whether the length of `x` is less than that of `y`.

Problem 1.23. Prove that `<` is transitive.

Problem 1.24. Define `(del e x)` to be the list obtained by deleting the first occurrence of `e` (as an element) from the list `x`, or `x` if `e` does not occur as an element of `x`.

Problem 1.25. Prove that `(mem e x)` implies that `(< (del e x) x)`.

Problem 1.26. Define `(perm a b)` to be `t` or `nil` according to whether the sequence of elements in list `a` is a permutation of the sequence of elements in list `b`.

Problem 1.27. Prove `(perm x x)`.

Problem 1.28. Prove that `(perm x y)` implies `(perm y x)`.

Problem 1.29. Prove that `perm` is transitive.

Problem 1.30. We say a sequence is *ordered* if successive element, e_i and e_{i+1} stand in the relation `(<` e_i e_{i+1}`)`. Define the predicate `ordered` to recognize ordered lists.

Problem 1.31. Define `(sort a)` so that it returns an ordered permutation of `a`.

References

[1] J. R. Shoenfield. *Mathematical Logic.* Addison-Wesley, Reading, Ma. 1967.