united nations
educational, scientific
and cultural
organization

international atomic
energy agency

the
abdus salam
international centre for theoretical physics

H4.SMR/1330-18

" Sixth Workshop on Non-Linear Dynamics and Earthquake Prediction"

15 - 27 October 2001

II. LISP: A Formalism for Expressing Mathematical Algorithms

H. Moritz

Graz Technical University
Institute of Physical Geodesy
Graz, Austria

II. LISP: A Formalism for Expressing Mathematical Algorithms

G. CHAITIN

(SEARCH FOR HIM IN THE INTERNET)

Synopsis

Why LISP?! S-expressions, lists & atoms. The empty list. Arithmetic in LISP. M-expressions. Recursive definitions. Factorial. Manipulating lists: car, cdr, cons. Conditional expressions: if-then-else, atom, =. Quote, display, eval. Length & size. Lambda expressions. Bindings and the environment. Let-be-in & define. Manipulating finite sets in LISP. LISP interpreter run with the exercises.

Why LISP?!

You might expect that a book on the incompleteness theorems would include material on symbolic logic and that it would present a formal axiomatic system, for example, Peano arithmetic. Well, this book won't! First, because formalisms for deduction failed and formalisms for computation succeeded. So instead I'll show you a beautiful and highly mathematical formalism for expressing algorithms, LISP. The other important reason for showing you LISP instead of a formal axiomatic system is that I can prove incompleteness results in a very general way without caring about the internal details of the formal axiomatic system. To me a formal axiomatic system is like a black box that theorems come out of. My methods for obtaining incompleteness theorems are so general that all I need to know is that there's a proof-checking algorithm.

So this is a post-modern book on incompleteness! If you look at another little book on incompleteness, Nagel & Newman's *Gödel's Proof,* which I enjoyed greatly as a child, you'll see plenty of material on symbolic logic, and **no** material on computing. In my opinion the right way to think about all this now is to start with computing, not with logic. So that's the way I'm going to do things here. There's no point in writing this book if I do everything the way everyone else does!

LISP means ``list processing,'' and was invented by John McCarthy and others at the MIT Artificial Intelligence Lab around 1960. You can do numerical calculations in LISP, but it's really intended for symbolic calculations. In fact LISP is really a computerized version of set theory, at least set theory for finite sets. It's a simple but very powerful mathematical formalism for expressing algorithms, and I'm going to use it in Chapters III, IV & V to present Gödel, Turing's and my work in detail. The idea is to start with a small number of powerful concepts, and get everything from that. So LISP, the way I do it, is more like mathematics than computing.

And LISP is rather different from ordinary work-a-day programming languages, because instead of executing or running programs you think about evaluating expressions. It's an expression-based or functional programming language rather than a statement-based imperative programming language.

Two versions of the LISP interpreter for this book are available at my web site: one is 300 lines of *Mathematica*, and the other is a thousand lines of *C*.

S-expressions, lists & atoms, the empty list

The basic concept in LISP is that of a *symbolic* or S-expression. What's an S-expression? Well, an S-expression is either an *atom*, which can either be a natural number like 345 or a word like Frederick, or it's a *list* of atoms or sub-lists. Here are three examples of S-expressions:

```
(), (a b c), (+ (* 3 4) (* 5 6))
```

The first example is the empty list (), also called *nil*. The second example is a list of three elements, the atoms *a*, *b* and *c*. And the third example is a list of three elements. The first is the atom +, and the second and the third are lists of three

elements, which are atoms. And you can nest lists to arbitrary depth. You can have lists of lists of lists...

The empty list () is the only list that's also an atom, it has no elements. It's indivisible, which is what atom means in Greek. It can't be broken into smaller pieces.

Elements of a list can be repeated, e.g., (a a a). And changing the order of the elements changes the list. So (a b c) is **not** the same as (c b a) even though it has the same elements, because they are in a different order. Hence *lists* are different from *sets*, where elements cannot be repeated and the order of the elements is irrelevant.

Now in LISP **everything**, programs, data, and output, they are all S-expressions. That's our universal substance, that's how we build everything!

Arithmetic in LISP

Let's look at some simple LISP expressions, expressions for doing arithmetic. For example, here is an arithmetic expression that adds the product of 3 and 4 to the product of 5 and 6:

3*4 + 5*6

The first step to convert this to LISP is to put in **all** the parentheses:

((3*4) + (5*6))

And the next step to convert this to LISP is to put the operators before the operands (prefix notation), rather than in the middle (infix notation):

(+(*34)(*56))

And now we need to use blanks to separate the elements of a list if parentheses don't do that for us. So the final result is

(+(* 3 4)(* 5 6))

which is already understandable, or

(+ (* 3 4) (* 5 6))

which is the standard LISP notation in which the successive elements of a list are always separated by a single blank. Extra blanks, if included, are ignored. Extra parentheses are **not** ignored, they completely change the meaning of an S-expression.

(((X))) is **not** the same as X!

What are the built-in operators, i.e., the primitive functions, that are provided for doing arithmetic in LISP? Well, in my LISP there's addition +, multiplication *, subtraction -, exponentiation ^, and, for comparisons, less-than <, greater-than >, equal =, less-than-or-equal <=, and greater-than-or-equal >=. I only provide natural numbers in my LISP, so there are no negative integers. If the result of a subtraction is less than zero, it gives 0 instead. Comparisons either give *true* or *false*. All these operators, or functions, always have two operands, or arguments. Addition and multiplication of more than two operands requires several + or * operations. Here are examples of arithmetic expressions in LISP, together with their values.

(+ 1 (+ 2 (+ 3 4))) is the same as (+ 1 (+ 2 7)) is the same as (+ 1 9) which yields 10,
(+ (+ 1 2) (+ 3 4)) is the same as (+ 3 7) which yields 10,
(- 10 7) yields 3,
(- 7 10) yields 0,
(+ (* 3 4) (* 5 6)) is the same as (+ 12 30) which yields 42,
(^ 2 10) yields 1024,
(< (* 10 10) 101) is the same as (< 100 101) which yields true,
(= (* 10 10) 101) is the same as (= 100 101) which yields false.

M-expressions

In addition to the official LISP S-expressions, there are also *meta* or M-expressions. These are just intended to be helpful, as a convenient abbreviation for the official S-expressions. Programmers usually write M-expressions, and then the LISP interpreter converts them into S-expressions before processing them. M-expressions omit some of the parentheses, the ones that group the primitive built-in functions together with their arguments. M-expression notation works because all built-in primitive functions in my LISP have a fixed-number of operands or arguments. Here are the M-expressions for the above examples.

+ 1 + 2 + 3 4 is the same as (+ 1 (+ 2 (+ 3 4))),
+ + 1 2 + 3 4 is the same as (+ (+ 1 2) (+ 3 4)),
- 10 7 is the same as (- 10 7),
- 7 10 is the same as (- 7 10),
+ * 3 4 * 5 6 is the same as (+ (* 3 4) (* 5 6)),
^ 2 10 is the same as (^ 2 10),
< * 10 10 101 is the same as (< (* 10 10) 101),
= * 10 10 101 is the same as (= (* 10 10) 101).

If you look at these examples you can convince yourself that there is never any ambiguity in restoring the missing parentheses, if you know how many arguments each operator has. So M-expressions are an example of what used to be called parenthesis-free or Polish notation. [This is in honor of a brilliant school of Polish logicians and set theorists.]

There are circumstances in which one wants to give parentheses explicitly rather than implicitly. So I use the notation " within an M-expression to indicate that what follows is an S-expression. Here are three examples:

The M-expression "(+ + +) denotes the S-expression (+ + +).
The M-expression ("+ "- "*) denotes the S-expression (+ - *).
The M-expression + "* "^ denotes the S-expression (+ * ^).

If one didn't use the double quotes, these arithmetic operators would have to have operands, but here they are just used as symbols. But these are not useful S-expressions, because they are not valid arithmetical expressions.

Recursive definitions & factorial

Now let me jump ahead and give the traditional first example of a complete LISP program. Later I'll explain everything in more detail. So here is how you define factorial in LISP. What's factorial? Well, the factorial of *N*, usually denoted *N*!, is the product of all natural numbers from 1 to *N*. So how do we do this in LISP? Well, we define factorial recursively:

(define (fact N) (if (= N 0) [then] 1 [else] (* N (fact (- N 1)))))

Here comments are enclosed in square brackets. This is the definition in the official S-expression notation. Here the definition is written in the more convenient M-expression notation.

define (fact N)
if = N 0 [then] 1
[else] * N (fact - N 1)

This can be interpreted unambiguously if you know that ``define" always has two operands, and ``if" always has three.

Let's state this definition in words.

The factorial function has one argument, *N*, and is defined as follows. If *N* is equal to 0 then factorial of *N* is equal to 1. Otherwise factorial of *N* is equal to *N* times factorial of *N* minus 1.

So using this definition, we see that (fact 4) is the same as

(* 4 (fact 3))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (* 2 (fact 1))))
(* 4 (* 3 (* 2 (* 1 (fact 0)))))
(* 4 (* 3 (* 2 (* 1 1))))
(* 4 (* 3 (* 2 1)))
(* 4 (* 3 2))
(* 4 6)

which yields 24.

So in LISP you define functions instead of indicating how to calculate them. The LISP interpreter's job is to figure out how to calculate them using their definitions. And instead of loops, in LISP you use recursive function definitions. In other words, the function that you are defining recurs in its own definition. The general idea is to reduce a complicated case of the function to a simpler case, etc., etc., until you get to cases where the answer is obvious.

Manipulating lists: car, cdr, cons

The examples I've given so far are all numerical arithmetic. But LISP is actually intended for symbolic processing. It's actually an arithmetic for lists, for breaking apart and putting together lists of things. So let me show you some list expressions instead of numerical expressions.

Car and *cdr* are the funny names of the operations for breaking a list into pieces. These names were chosen for historical reasons, and no longer make any sense, but everyone knows them. If one could start over you'd call car, head or first and you'd call cdr, tail or rest. Car returns the first element of a list, and cdr returns what's left without the first element. And *cons* is the inverse operation, it joins a head to a tail, it adds an element to the beginning of a list. Here are some examples, written in S-expression notation:

(car (' (a b c))) yields a,
(cdr (' (a b c))) yields (b c),
(car (' ((a) (b) (c)))) yields (a),
(cdr (' ((a) (b) (c)))) yields ((b) (c)),
(car (' (a))) yields a,
(cdr (' (a))) yields (),
(cons (' a) (' (b c))) yields (a b c),
(cons (' (a)) (' ((b) (c)))) yields ((a) (b) (c)),
(cons (' a) (' ())) yields (a),
(cons (' a) nil) yields (a),
(cons a nil) yields (a).

Maybe these are easier to understand in M-expression notation:

car '(a b c) yields a,
cdr '(a b c) yields (b c),
car '((a) (b) (c)) yields (a),
cdr '((a) (b) (c)) yields ((b) (c)),
car '(a) yields a,
cdr '(a) yields (),
cons 'a '(b c) yields (a b c),
cons '(a) '((b) (c)) yields ((a) (b) (c)),
cons 'a '() yields (a),
cons 'a nil yields (a),
cons a nil yields (a).

There are some things to explain here. What is the single quote function? Well, it just indicates that its operand is data, not an expression to be evaluated. In other words, single quote means ``literally this.'' And you also see that the value of nil is (), it's a friendlier way to name the empty list. Also, in the last example, *a* isn't quoted, because initially all atoms evaluate to themselves, are bound to themselves. Except for nil, which has the value (), every atom gives itself as its value initially. This will change if an atom is used as the parameter of a function and is bound to the value of an argument of the function. Numbers though, **always** give themselves as value.

Here are some more examples, this time in M-expression notation. Note that the operations are done from the inside out.

car '(1 2 3 4 5) yields 1,
car cdr '(1 2 3 4 5) yields 2,
car cdr cdr '(1 2 3 4 5) yields 3,
car cdr cdr cdr '(1 2 3 4 5) yields 4,
car cdr cdr cdr cdr '(1 2 3 4 5) yields 5,
cons 1 cons 2 cons 3 cons 4 cons 5 nil yields (1 2 3 4 5).

This is how to get the second, third, fourth and fifth elements of a list. These operations are so frequent in LISP that they are usually abbreviated as cadr, caddr, cadddr, caddddr, etc., and so on and so forth with all possible combinations of car and cdr. My LISP though, only provides the first two of these abbreviations, cadr and caddr, for the second and third elements of a list.

Conditional expressions: if-then-else, atom, =

Now I'll give a detailed explanation of the three-argument function if-then-else that we used in the definition of factorial.

(if predicate then-value else-value)

This is a way to use a logical condition, or predicate, to choose between two alternative values. In other words, it's a way to define a function by cases. If the predicate is true, then the then-value is evaluated, and if the predicate is false, then the else-value is evaluated. The unselected value is **not** evaluated. So if-then-else and single quote are unusual in that they do not evaluate all their arguments. Single quote never evaluates its one argument, and if-then-else only evaluates two of its three arguments. So these are pseudo-functions, they are not really normal functions, even though they are written the same way that normal functions are.

And what does one use as a predicate for if-then-else? Well, for numerical work one has the numerical comparison operators <, >, <=, >=, and =. But for work with S-expressions there are just two predicates, *atom* and =. Atom returns true or false depending upon whether its one argument is an atom or not. = returns true or false depending upon whether two S-expressions are identical or not. Here are some examples written in S-expression notation:

(if (= 10 10) abc def) yields abc,
(if (= 10 20) abc def) yields def,
(if (atom nil) 777 888) yields 777,
(if (atom (cons a nil)) 777 888) yields 888,
(if (= a a) X Y) yields X,
(if (= a b) X Y) yields Y.

Quote, display, eval

So, to repeat, single quote never evaluates its argument. Single quote indicates that its operand is **data**, not an expression to be evaluated. In other words, single quote means ``literally this.'' Double quote is for including S-expressions inside of M-expressions. Double quote indicates that parentheses will be given explicitly in this part of an M-expression. For example, the M-expressions '+ 10 10 and ' "(+ 10 10) and '("+ 10 10) all denote the S-expression (' (+ 10 10)), which yields value (+ 10 10), **not** 20.

Display, which is useful for obtaining intermediate values in addition to the final value, is just an identity function. Its value is the same as the value of its argument, but it has the side-effect of displaying its argument. Here is an example written in M-expression notation:

car display cdr display cdr display cdr '(1 2 3 4 5)
displays (2 3 4 5)
displays (3 4 5)
displays (4 5)
and yields value 4.

Eval provides a way of doing a **stand-alone** evaluation of an expression that one has constructed. For example:

eval display cons "^ cons 2 cons 10 nil
displays (^ 2 10)
and yields value 1024.

This works because LISP is interpreted, not compiled. Instead of translating a LISP expression into machine language and then running it, the LISP interpreter is always present doing evaluations and printing results. It is very important that the argument of eval is always evaluated in a clean, initial environment, not in the current environment. I.e., the only binding in effect will be that nil is bound to (). All other atoms are bound to themselves. In other words, the expression being evaled must be self-contained. This is important because in this way the result of an eval doesn't depend on the circumstances when it is used. It always gives the same value if it is given the same argument.

Length & size

Here are two ways to measure how big an S-expression is. *Length* returns the number of elements in a list, i.e., at the top level of an S-expression. And *size* gives the number of characters in an S-expression when it is written in standard notation, i.e., with exactly one blank separating successive elements of a list. For example, in M-expression notation:

length '(a b c) yields 3,
size '(a b c) yields 7.

Note that the size of (a b c) is 7, not 8, because when size evaluates its argument the single quote disappears. [If it didn't, its size would be the size of (' (a b c)), which is 11!]

Lambda expressions

Here is how to define a function.

(lambda (list-of-parameter-names) function-body)

For example, here is the function that forms a pair in reverse order.

(lambda (x y) (cons y (cons x nil)))

Functions can be literally given in place (here I've switched to M-expression notation):

('lambda (x y) cons y cons x nil A B) yields (B A)

Or, if *f* is bound to the above function definition, then you can use it like this

(f A B) yields (B A)

(The general idea is that the function is always evaluated before its arguments are, then the parameters are bound to the argument values, and then the function body is evaluated in this new environment.) How can we bind *f* to this function definition? Here's a way that's permanent.

define (f x y) cons y cons x nil

Then (f A B) yields (B A). And here's a way that's local.

('lambda (f) (f A B) 'lambda (x y) cons y cons x nil)

This yields (B A) too. If you can understand this example, then you understand all of my LISP! It's a list with two

elements, the expression to be evaluated that uses the function, and the function's definition. Here's factorial done the same way:

('lambda (fact) (fact 4) 'lambda (N) if = display N 0 1 * N (fact - N 1))

This displays 4, 3, 2, 1, and 0, and then yields the value 24. Please try to understand this final example, because it **really** shows how my LISP works!

Bindings and the environment

Now let's look at defined (as opposed to primitive) functions more carefully. To define a function, one gives a name to the value of each of its arguments. These names are called parameters. And then one indicates the body of the function, i.e., how to compute the final value. So within the scope of a function definition the parameters are bound to the values of the arguments.

Here is an example, in which a function returns one of its arguments without doing anything to it. ('lambda (x y) x 1 2) yields 1, and ('lambda (x y) y 1 2) yields 2. Why? Because inside the function definition *x* is bound to 1 and *y* is bound to 2. But all previous bindings are still in effect except for bindings for *x* and *y*. And, as I've said before, in the initial, clean environment every atom except for nil is bound to itself. Nil is bound to (). Also, the initial bindings for numbers can never be changed, so that they are constants.

Let-be-in & define

Local bindings of functions and variables are so common, that we introduce an abbreviation for the lambda expressions that achieve this. To bind a variable to a value we write:

(let variable [be] value [in] expression)

And to bind a function name to its definition we write it like this.

(let (function-name parameter1 parameter2...) [be] function-body [in] expression)

For example (and now I've switched to M-expression notation)

let x 1 let y 2 + x y yields 3.

And

let (fact N) if = N 0 1 * N (fact - N 1) (fact 4) yields 24.

Define actually has two cases like let-be-in, one for defining variables and another for defining functions:

(define variable [to be] value)

and

(define (function-name parameter1 parameter2...) [to be] function-body)

The scope of a define is from the point of definition until the end of the LISP interpreter run, or until a redefinition occurs. For example:

define x 1
define y 2
Then + x y yields 3.

define (fact N) if = N 0 1 * N (fact - N 1)
Then (fact 4) yields 24.

Define can only be used at the ``top level." You are not allowed to include a define inside a larger S-expression. In other words, define is not really in my LISP. Actually all bindings are local and should be done with let-be-in, i.e., with lambda bindings. Like M-expressions, define and let-be-in are a convenience for the programmer, but they're not officially in my LISP, which only allows lambda expressions and temporary local bindings. Why? **Because in theory each LISP expression is supposed to be totally self-contained, with all the definitions that it needs made locally!** Why? Because that way the size of the smallest expression that has a given value, i.e., the LISP program-size complexity, does not depend on the environment.

III. Gödel's Proof of his Incompleteness Theorem

Synopsis

Discusses a LISP run exhibiting a fixed point, and a LISP run which illustrates Gödel's proof of his incompleteness theorem.

Fixed points, self-reference & self-reproduction

Okay, let's put LISP to work! First let me show you the trick at the heart of Gödel's and Turing's proofs, the self-reference.

How can we enable a LISP expression to know itself? Well, it's very easy once you've seen the trick! Consider the LISP function f(x) that takes x into (('x)('x)). In other words, f assumes that its argument x is the lambda expression for a one-argument function, and it forms the expression that applies x to x. It doesn't evaluate it, it just creates it. It doesn't actually apply x to x, it just creates the expression that will do it.

You'll note that if we **were** to evaluate this expression (('x)('x)), in it x is simultaneously program and data, active and passive.

Okay, so let's pick a particular x, use f to make x into (('x)('x)), and then run/evaluate the result! And to which x shall we apply f? Why, to f itself!

So f applied to f yields what? It yields f applied to f, which is what we started with!! So f applied to f is a self-reproducing LISP expression!

You can think of the first f, the one that's used as a function, as the organism, and the second f, the one that's copied twice, that's the genome. In other words, the first f is an organism, and the second f is its DNA! I think that that's the best way to remember this, by thinking it's biology. Just as in biology, where a organism cannot copy itself directly but needs to contain a description of itself, the self-reproducing function f cannot copy itself directly (because it cannot read itself---and neither can **you**). So f needs to be given a (passive) copy of itself. The biological metaphor is quite accurate!

So in the next section I'll show you a LISP run where this actually works. There's just one complication, which is that what can reproduce itself, in LISP just as in real life, depends on the environment in which the organism finds itself.

(f f) works in an environment that has a definition for f, but that's cheating! What I want is a **stand-alone** LISP expression that reproduces itself. But f produces a stand-alone version of itself, and **that** is the actual self-reproducing expression. (f f) is like a virus that works only in the right environment (namely in the cell that it infects), because it's too simple to work on its own.

So, finally, here is the LISP run illustrating all this. I hope you like it, and that it convinces you that it was worth the effort to learn LISP! After you understand this LISP run, I'll show you Gödel's proof.

A LISP Fixed Point

```
LISP Interpreter Run

[[[[[

 A LISP expression that evaluates to itself!

 Let f(x): x -> (('x)('x))
```

```
 Then (('f)('f)) is a fixed point.

]]]]]

[Here is the fixed point done by hand:]

(
'lambda(x) cons cons "' cons x nil
           cons cons "' cons x nil
                nil

'lambda(x) cons cons "' cons x nil
           cons cons "' cons x nil
                nil
)

expression  ((' (lambda (x) (cons (cons ' (cons x nil)) (cons
            (cons ' (cons x nil)) nil)))) (' (lambda (x) (cons
             (cons ' (cons x nil)) (cons (cons ' (cons x nil))
             nil)))))
value       ((' (lambda (x) (cons (cons ' (cons x nil)) (cons
            (cons ' (cons x nil)) nil)))) (' (lambda (x) (cons
             (cons ' (cons x nil)) (cons (cons ' (cons x nil))
             nil)))))


[Now let's construct the fixed point.]

define (f x) let y [be] cons "' cons x nil  [ y is ('x)          ]
             [return] cons y cons y nil     [ return (('x)('x)) ]

define      f
value       (lambda (x) ((' (lambda (y) (cons y (cons y nil)))
            ) (cons ' (cons x nil))))


[Here we try f:]

(f x)

expression  (f x)
value       (('  x) (' x))


[Here we use f to calculate the fixed point:]

(f f)

expression  (f f)
value       ((' (lambda (x) ((' (lambda (y) (cons y (cons y ni
            l)))) (cons ' (cons x nil))))) (' (lambda (x) (('
            (lambda (y) (cons y (cons y nil)))) (cons ' (cons
            x nil))))))


[Here we find the value of the fixed point:]

eval (f f)

expression  (eval (f f))
value       ((' (lambda (x) ((' (lambda (y) (cons y (cons y ni
            l)))) (cons ' (cons x nil))))) (' (lambda (x) (('
            (lambda (y) (cons y (cons y nil)))) (cons ' (cons
            x nil))))))


[Here we check that it's a fixed point:]

= (f f) eval (f f)
```

```
expression  (= (f f) (eval (f f)))
value       true


[Just for emphasis:]

= (f f) eval eval eval eval eval eval (f f)

expression  (= (f f) (eval (eval (eval (eval (eval (eval (f f)
            )))))))
value       true

End of LISP Run

Elapsed time is 0 seconds.
```

Metamathematics in LISP

Now on to Gödel's proof!

Turing's proof and mine do not depend on the inner structure of the formal axiomatic system being studied, but Gödel's proof does. He needs to get his hands dirty, he needs to lift the hood of the car and poke around in the engine! In fact, what he needs to do is to **confuse levels** and combine the theory and its metatheory. That's how he can construct a statement in the theory that says that it's unprovable.

In Gödel's original proof, he was working in Peano arithmetic, which is just a formal axiomatic system for elementary number theory. It's the theory for the natural numbers and plus, times, and equal. So Gödel used Gödel numbering to **arithmetize metamathematics**, to construct a numerical predicate Dem(p,t) that is true iff p is the number of a proof and t is the number of the theorem that p proves.

Gödel did it, but it was hard work, very hard work! So instead, I'm going to use LISP. We'll need a way to express metamathematical assertions using S-expressions. We'll need to use S-expressions to express proofs and theorems. We need to construct a LISP function (valid-proof? x) that returns the empty list nil if x is not a valid proof, and that returns the S-expression for the theorem that was demonstrated if x is a valid proof. I won't actually define/program out the LISP function valid-proof?. I don't want to get involved ``in the internal affairs" of a particular formal axiomatic system. But you can see that it's not difficult. Why not?

Well, that's because Gödel numbers are a difficult way to express proofs, but S-expressions are a very natural way to do it. An S-expression is just a symbolic expression with explicit syntax, with its structure completely indicated by the parentheses. And it's easy to write proof-checking algorithms using LISP functions. LISP is a natural language in which to express such algorithms.

So let's suppose that we have a definition for a LISP function (valid-proof? x). And let's assume that the formal axiomatic system that we're studying is one in which you can talk about S-expressions and the value of an S-expression. Then how can we construct a LISP expression that asserts that it's unprovable?

First of all, how can we state that an S-expression y is unprovable? Well, it's just the statement that for all S-expressions x, it is not the case that (valid-proof? x) is equal to y. So that's easy to do. So let's call the predicate that affirms this is-unprovable. Then what we need is this: a LISP expression of the form (is-unprovable (value-of XXX)), and when you evaluate the LISP expression XXX, it gives back this entire expression. (value-of might more accurately be called lisp-value-of, just to make the point that we are assuming that we can talk about LISP expressions and their values in our formal axiomatic system.)

So we're almost there, because in order to do this we just need to use the fixed-point trick from before in a slightly more complicated manner. Here's how:

Gödel's Proof in LISP

```
LISP Interpreter Run

[[[[[

 A LISP expression that asserts that it itself is unprovable!

 Let g(x): x -> (is-unprovable (value-of (('x)('x))))

 Then (is-unprovable (value-of (('g)('g))))
 asserts that it itself is not a theorem!

]]]]]

define (g x)
   let (L x y) cons x cons y nil [Makes x and y into list.]
   (L is-unprovable (L value-of (L (L "' x) (L "' x))))

define      g
value       (lambda (x) ((' (lambda (L) (L is-unprovable (L va
            lue-of (L (L ' x) (L ' x)))))) (' (lambda (x y) (c
            ons x (cons y nil))))))


[Here we try g:]

(g x)

expression  (g x)
value       (is-unprovable (value-of ((' x) (' x))))


[
 Here we calculate the LISP expression
 that asserts its own unprovability:
]

(g g)

expression  (g g)
value       (is-unprovable (value-of ((' (lambda (x) ((' (lamb
            da (L) (L is-unprovable (L value-of (L (L ' x) (L
            ' x)))))) (' (lambda (x y) (cons x (cons y nil))))
            ))) (' (lambda (x) ((' (lambda (L) (L is-unprovabl
            e (L value-of (L (L ' x) (L ' x)))))) (' (lambda (
            x y) (cons x (cons y nil)))))))))))


[Here we extract the part that it uses to name itself:]

cadr cadr (g g)

expression  (car (cdr (car (cdr (g g)))))
value       ((' (lambda (x) ((' (lambda (L) (L is-unprovable (
            L value-of (L (L ' x) (L ' x)))))) (' (lambda (x y
            ) (cons x (cons y nil))))))) (' (lambda (x) ((' (l
            ambda (L) (L is-unprovable (L value-of (L (L ' x)
            (L ' x)))))) (' (lambda (x y) (cons x (cons y nil)
            )))))))


[Here we evaluate the name to get back the entire expression:]

eval cadr cadr (g g)

expression  (eval (car (cdr (car (cdr (g g))))))
value       (is-unprovable (value-of ((' (lambda (x) ((' (lamb
            da (L) (L is-unprovable (L value-of (L (L ' x) (L
            ' x)))))) (' (lambda (x y) (cons x (cons y nil))))
            ))) (' (lambda (x) ((' (lambda (L) (L is-unprovabl
            e (L value-of (L (L ' x) (L ' x)))))) (' (lambda (
            x y) (cons x (cons y nil)))))))))))
```

```
[Here we check that it worked:]

= (g g) eval cadr cadr (g g)

expression  (= (g g) (eval (car (cdr (car (cdr (g g)))))))
value       true

End of LISP Run

Elapsed time is 0 seconds.
```
