# An Interpreter for a LISP-like Language

**Lewis Smith**
8933715

*University of Manchester*

*Any sufficiently complex C or FORTRAN program contains a buggy, inconsistent, ad hoc, informally specified, bug-ridden, slow implementation of half of common lisp.*

*- Philip Greenspun*

## Abstract

This report details the design and implementation of a minimal interpreter for a language in the LISP family, using C++. The object oriented features of C++ are used to express the datatypes and primitives of the language in an elegant and self contained way. Since a large section of the functionality of an interpreter must be determined at runtime by the interactions with the user, extensive use must be made of runtime memory management and polymorphism, which is made clearer and simpler via effective class design and seperation¿

## 1 Introduction

An *interpreter* is a program that directly executes a series of commands, transforming instructions in the form of source code in a specified language into computer actions. 'Real world' interpreters for production languages are generally programs of great complexity, however if performance is not a priority a demonstration of the core functionality is tractable. Lisp was chosen as the target for this interpreter due to the conceptual simpicity of it's syntax and semantics, which greatly simplifies the implementation. All of lisps core syntax can be expressed as follows:

```
(function argument1 argument2 ...)
```

which translates to 'apply function to arg1, arg2, arg3...'. This makes lisp look slightly strange [1] but it greatly simplifies the implementation of the parser, making it plausible to write one by hand rather than using automatic parser generation software such as lexx and yacc, which are generally used for languages with more complex grammar. Classical lisp also makes use of only a simple compund datatype, the hetereogenous linked list, which also serves as the representation of the lanuage itself. Therefore, the entire language, both the representation of the code and it's execution, are facilitated by the implementation of a single data structure, which is in the c++ standard library (std::list).

The functionality of an interpreter can be broken down into a few components, which can be implemented as seperated classes:

1. The representation of language constructs as data structures. Evaluation of the language can be implemented as a method of this class, due to the equivalence between lisp code and data
2. The parser, which translates the string representation of a program into the internal representation of that program.
3. The environment, which allows the binding of language values to names
4. The garbage collector, which is responsible for managing the memory usage of the lisp environment.

[1] Lisp was gets it's name from a contraction of 'List Processing', but there is an old joke that claims it stands for 'Lots of Irritating Sets of Parenthesis'

# 2 Design

## 2.1 S-Expressions

Lisp code consists of 'S-Expressions', or symbolic expressions. As briefly discussed earlier, s-expressions are defined by the following rules:

1. An atom, or symbolic identifier of the language, is an s-expression. For example, the name

   ```
   x
   ```

   is an s-expression.

2. A number or string literal is an s-expression. For example, both of the following are individual s-expressions

   ```
   "hello" 3.14159
   ```

3. A list whose elements are s-expressions is an s-expression. Lists are written as space-seperated elements enclosed in parentheses. This definition is recursive, so arbitrarily nested lists are s-expressions:

   ```
   ("hello" x 3.14159 (a b 27))
   ```

We can find the *value* of a lisp statement by it's literal value if it is a string or number, by looking up it's definition in a symbol table if it is an atom, or by evaluating it as a function call, with the first element as the function and the rest of the list as the argument as mentioned above. If the expression is ill-formed, for example if the first element of the list is not a function, or an atom is not definedwhen the expression is called, the the interpreter should respond by printing an error message, or similar.

We can express this concept in C++ in a natural way by defining a *interface* that describes the behaviour of an s-expression, via an abstract class. The conceptual sketch of this class is as follows:

```
class SExp {
public:
  virtual SExp *eval(Env &env) = 0;
};
```

That is, an s-expression object must be able to take a reference to the current environment, and return a pointer to an s-expression representing it's value. This could be simply a pointer to itself if it is literal, or a pointer to a newly created object if it is a function call. All instances of s-expressions can then inherit from this abstract class, overriding the eval method, which allows the runtime determination of the evaluation rules based on the type of object which is being evaluated. Polymorphism is also neccesary in order to be able to store s-expressions in a container, which is required to implement the list class, since C++ only allows objects of the same type to be stored in data structures. Having defined an abstract class that all lisp objects will satisfy, sketching the implementation of a list class is easy, if the STL is used:

```
class List : public SExp {
public:
  List(std::list<SExp* > list) : elems(list) {}
  const std::list<SExp *> elems;
  virtual SExp *eval(Env &env) override;
};
```

## 2.2 Lexing and Parsing

The task of transforming a stream of characters into the data structures described above can be further split into two components, the *lexer* and the *parser*. The lexer transforms an input string into a sequence of tokens, while the parser consumes the token stream and constructs the s-expression objects. For example, the lexer will transform an input string like the following

```
(1 "hello" 3 6 func)
```

into a sequence of tokens, optionally accompanied by the values of the literal that was read.

[open-bracket], [number 1], [string "hello"], [number 3], [number 6], [atom 'func'], [close-bracket], [end]

In practice, the lexer will take a reference to a stream to read from, and then implement public functions to return the next token in the stream, and to read the last number or string encountered by the lexer. This class can then be embedded in the parser, making use of composition.

## 2.3 Environment and Scopes

At it's simplest, the environment class is simply a wrapper around a table that associated strings, the names of the symbols defined in the language, with pointers to Lisp objects. The standard data structure for this kind of lookup table is a *hash table* or *map*. Similarly to lists, there is no need to define this from scratch since there is an implementation in the STL. The environment class must provide functions that allow the binding of new variables to the namespace and the accessing of variables already defined.

## 2.4 Garbage Collection

Lisp is a garbage collected language, meaning that the responsibility for manually managing heap memory is delegated to the language runtime, rather than being the responsibility of the programmer as in C++. As such, it is neccesary to manually implement a garbage collector as part of the lisp runtime. This must track the usage of memory, detecting when a heap block is no longer referenced by anything in the lisp environment and cleaning up such memory as needed. An object is 'in use' by lisp if it is reachable from the symbol table, directly or indirectly: if no way to reach it exists then it is garbage.

```
((lambda (x) (define a 3) (+ x a)) 2)
;;returns 5. 'a' was defined in the funtions local scope,
;; but is now garbage, since it is not reachable from the global scope
```

One option for a garbage-collection like facility that is provided in C++ 11 is reference counting, where an pointer to a object maintains a record of how many other pointers that are also pointing to the same memory exist in the program, deallocating them when the refcount reaches zero. This is implemented by C++'1 shared pointer class.

Unfortunately, reference counting is not a complete form of garbage collection. In particular, it is able to leak memory if cirular references are possible: that is, if object A contains a pointer