Lpp User's Manual

Table of Contents

1 Introduction

1.1 Overview

Lpp (Lisp Plus Plus), is a library of Lisp like functions and macros usable in C++ programs. The philosophy behind Lpp is to provide as close as possible the semantics and style of Lisp rather than try to force it to fit a static style of programming. Lpp tries to emulate Common Lisp as much as possible in this regard. By doing things this way part of the true power and flexibility of Lisp can coexist and mix with the static typing features of C++ even within functions and objects. The hope is that Lpp will be useful for the following

- Porting Lisp programs to and from C++.
- Implementation of embedded AI sub-systems in C++ environments.
- An alternative for Lisp programmers who need to program in C++.
- When a C++ program needs dynamically typed objects.
- When a C++ program needs an efficient unified list implementation.
- When a C++ program needs rational numbers in the range of minus to plus infinity.
- When a C++ program needs to do symbolic processing such as manipulating s-expressions, symbolic math, etc ...

One of Lisp's advertised benefits is that of dynamically typed objects. Standard C++ does not offer this capability. Instead, the programmer is expected to created virtual functions whose objects dynamically dispatch through the use of indirect pointers to v-tables and then a v-table has a pointer to the virtual function for that object. While the basic idea of virtual functions is good, relying only on it for real world complex problems presents difficulties. One such problem is that it is impossible to write true generic code using virtual functions since all possible types must be accounted for. The default virtual function of a base class can only serve the declared sub-classes that do not define their own virtual functions. Contrast this with Lisp functions that usually do not have to account for the types of the objects that it operates on. As a very simple example, Lisp can compute the length of a list irrespective of the types of the objects in the list. Furthermore a provider could supply an object dynamically to a consumer of such a list that is not required to be seen by the compiler of the program operating on the list.

Lpp strives to provide the full power of Lisp in terms of dynamic typing. The way that Lpp achieves this is that all Lpp objects contain a type header that is a pointer to a type meta-object. In some sense this is similar to the v-table concept mentioned above for virtual function tables in that type meta-objects can and do contain type dispatcher functions that are similar to virtual functions. But the type meta-objects are automatic in Lpp as well as much of their standard behavior. And type meta-objects are first class Lpp objects so that they can be manipulated dynamically as ordinary objects can. For example a dispatcher function can be dynamically added to a type meta-object and then operate on subsequent ordinary objects of that type. Furthermore the type meta-objects maintain a run time type lattice. For example at run time a program can query whether an object is a number or an integer (integers being a subset of all numbers).

All Lpp objects as in Lisp have the same base type. In Lisp the base type is type `t`. In Lpp the base type is intentionally left unknown and abstracted with a type definition called

`let`. This was done so that the base type can be experimented with while preserving Lpp syntax. Type `let` serves as the declaration type for all such Lpp objects when referring to the Lpp object as a generic object. As in Lisp Lpp objects are passed around as pointers with pointer abstraction in Lpp function arguments and Lpp slots. These objects can be mixed freely with other C++ classes and class members and can be used with other C++ libraries that do not use the same class names that Lpp uses.

Lpp provides a `the` macro for descending into specific Lpp objects. This is similar to the `the` special form in Common Lisp. Lpp automatically does a dynamic type check on the object wrapped in a `the` macro which assures that specific object member references will always be correct. In a debugged program compilation unit this type checking can be turned off for maximum compilation efficiency.

Lpp also provides other useful Lisp objects such as Cons cells, lists, symbols, strings, characters and numbers. As much as possible Lpp tries to use the Common Lisp function names and semantics for the operations on Lisp like objects.

Lpp chose address 0 as `nil`. This is so that it is easy to check for the end of lists or `nil` returned in predicate arguments in C++ functions, operators and statements like `if`, `while`, `for`. So `nil` and non-`nil` is consistent with C++ predicate semantics on 0 and non-0. But even though `nil` is 0, it is exactly equivalent to the symbol `nil`. It is as if the symbol object `nil` was an object at address 0.

Symbols are introduced into C++ programs easily with the `S()` macro. For example, `S(foo)`, `S(Foo)`, `S(my:foo)`, `S(foo-bar)`, `S(+)`, `S(!\@#$%^&*)` are all legitimate symbols in Lpp. Symbols after being interned are just as efficient but far more powerful than `enums`. The default for symbols in Lpp is case sensitive, so that `foo` and `Foo` are two distinct symbols.

Primitive C objects can easily be converted to their analog Lpp objects using the `L` constructor. For example `L("abc")` will produce an Lpp String object and `L(cdr)` will produce an Lpp first class function object.

Conversions back are also easy for example `22 == iL(L(22))` where iL conversion is back to C++ integers. The programmer can use dynamic Lpp objects when needed and use simple C++ objects when absolute efficiency is needed. Efficiency is an elusive property though. For example Lpp strings seem less efficient on the surface, but since they have a length they provide for faster string comparisons than ordinary 0 terminated C++ strings. The names of such things as `L` (the conversion macro), `True` and `S` (the symbol macro) can be redefined per compilation unit by the programmer without affecting the operation of Lpp.

Lpp provides the basic Common Lisp I/O functions like `read`, `print`, `prin1`, `princ`, `pprint` ... etc, but also provides that any Lpp object can appear in stream operators. So if `obj` is the Lpp symbol object `car` then `cout << "obj = " << obj` would print as `obj = car`. All Lpp objects have C++ stream print methods that inherit from the basic `princ` method of an object. All new Lpp objects defined automatically get default dispatcher functions defined for `princ` and `prin1`. New `princ` or `prin1` methods can be set dynamically by the programmer for Lpp type meta-objects.

Lpp provides mathematically correct rational numbers, ie. ratios whose numerator and denominator and integers are in the range minus to plus infinity. Lpp numbers automatically expand or shrink to any size when operated on by Lpp math functions. Overflow or underflow are impossible.

Lpp provides two debugger functions `pdb` and `pdc` for examining Lpp objects in a debugger. `pdb` uses `prin1` and `pdc` uses `princ`. So while in a debugger the user can type, for example, `p pdc(list1)` where `list1` is some variable in a program that contains an Lpp list, the list and all of it's subcomponents will be printed exactly as you would expect in a lisp interpreter. In addition the function `ppdb` is provided for pretty printing such objects in a debugger.

1.2 Installing Lpp

Lpp comes in a Gnu Tar Zipped file `lpp-VER.tgz` where `VER` represents the version of Lpp. See your system documentation on how to unpack it. If we represent the directory in which you unpacked as `UDIR` then after unpacking there will be a directory `UDIR/lpp`. To make the Lpp library and accompaniments first go to the `UDIR/lpp` directory and type

```
./configure
```

followed by

```
make
```

If both complete with no errors about missing components on your system then you are ready to install Lpp. If you are installing in the default or a protected area you probably need to log in as a super user. Then while in `UDIR/lpp` type

```
make install
```

If we refer to the default system installation directory as `IDIR`, on Linux for example `IDIR` will default to `/usr/local`, then the following will get installed in Linux subdirectories as follows

```
IDIR/doc/lpp/lpp.html
IDIR/include/lpp/Lpp.hh
IDIR/include/lpp/LppPure.hh
IDIR/include/lpp/rpMenu.hh
IDIR/info/lpp.info
IDIR/lib/lpp/libLpp.a
IDIR/lib/lpp/librpMenu.a
```

The file `UDIR/lpp/doc/INSTALL` has a more general description of `configure` and `make`. For example if you want to install in other than the default system installation directory then you can specify to install under some other directory `IDIR` as follows

```
./configure --prefix IDIR
make
```

At this point while still in the `UDIR` directory you can install the additional "No Tyoe Check" version of the Lpp library with the following

```
make nc
make installnc
```

and this, using our above example, will install

```
IDIR/lib/lpp/libLppNC.a
```

The Lpp User's Manual will explain when you might want to use `libLppNC`, See Section 3.6 [Type Checking], page 21. It doesn't hurt to do this and you may as well at this point.

This make will also create an Lpp regression test. If you want to see if it passes this test on your system you can do the following: Change to the directory `UDIR/lpp/lib/test` and type

```
check
```

It runs the regression test and then does a `diff` against the expected results. If you see

```
---- Lpp test suite #1 - difference between now and expected is:
```

and *no* following differences then it has passed the test on your system.

After the `make install`, referring to the above install directory example, you can then read the Lpp User's Manual using Emacs Info with the file `lpp.info` or as HTML using the URL

```
file:IDIR/doc/lpp/lpp.html
or
file:IDIR/doc/lpp/lpp.html/index.html
```

In particular you can read how to use Lpp, See Section 1.3 [Using Lpp], page 6.

If you want a hard copy of the Lpp User's Manual then change to the directory `UDIR/lpp/doc` and type `make ps` which will make the PostScript version `lpp.ps`.

1.3 Using Lpp

Using the Lpp library is easy. First you need to include in your C++ file

```
#include <Lpp.hh>
```

In some fussy cases you may want to include `LppPure.hh` here instead of `Lpp.h`, but `Lpp.hh` is always safe to use, See Section 1.5 [Pure Lisp Include File], page 10.

Second, you need to link the library `libLpp.a` with the compile of your C++ program. `libLpp.a` gets installed on your system when Lpp is installed, See Section 1.2 [Installing Lpp], page 4. For example on Linux if you had installed Lpp under `/usr/local` and then wrote a C++ program that used Lpp called `main.cc` then the following would first compile a program `main.cc` and then make an executable called `pgm`

```
c++ -I/usr/local/include -c main.cc
c++ -o pgm  main.o /usr/local/lib/lpp/libLpp.a
```

Then after that, if you are familiar with Common Lisp, you will find that writing programs using Lpp is similar to writing programs in Common Lisp. A big advantage is that you can mix quite freely the static style of C++ with the dynamic style of Lisp without having to think about it too much. In the next section we give a simple example to illustrate this.

1.4 A Simple Data Base Example

In this example we create a class that implements a simple data base. We then create an instance of the data base and populate it with some entities, namely some people. And finally we generate a report on the people of a family that we used to populate the data base to show the relational aspect of the data base.

Lpp is useful here since we want entities to be any kind of object and hence dynamically bound objects will index the data base. In the example we use Lpp symbols to designate entities. We set relations in the data base with a typical entity attribute value tuple. So the data base access member functions are as follows

- `getSize`() = Returns the total number of entities in the data base
- `addEntity`(*entity*) = Add *entity* (any Lpp object) to the data base
- `setValue`(*entity*, *attribute*, *value*) = Add a relation for *entity*
- `getValue`(`entity`, `attribute`) = Returns the *attribute* value for *entity*

Here is the implementation of the data base class

```
////////////////////////////////////////////////////////////
// main.cc = Introduction Simple Data Base example.

#include <Lpp.hh>

// Data Base class.
class DataBase {
  int size;
  let contents;
public:
  DataBase();
  int getSize() {return size;}
  void addEntity(let);
  void setValue(let, let, let);
  let getValue(let, let);};

// Data Base constructor.
DataBase::DataBase() {size = 0; contents = makeHashTable();}

// Add an entity to the Data Base.
void DataBase::addEntity(let entity) {
  if (!gethash(entity, contents)) {
    puthash(entity, contents, 0);
    size++;}}

// Set the value of an attribute for given entity.
void DataBase::setValue(let entity, let attribute, let value) {
  let attributes = gethash(entity, contents);
  let old = assoc(attribute, attributes);
  if (old) rplacd(old, value);
  else {
    push(cons(attribute, value), attributes);
    puthash(entity, contents, attributes);}}

// Return the value of an attribute for given entity.
let DataBase::getValue(let entity, let attribute) {
  return cdr(assoc(attribute, gethash(entity, contents)));}
```

First notice the `include` command for `Lpp.hh`. To use Lpp a compilation unit needs to `include` the header file `Lpp.hh` when using `let` declarations.

Notice in the class definition the `let` type declarations. This is how members containing Lpp objects are always declared. Also notice that `let` type members can be easily combined with other types, in this case `int`.

In the data base accessor functions and in the following code that uses them the occurrences of Common Lisp function names such as `car`, `cdr`, `cons`, `assoc`, `push`, `puthash`,

`dolist` etc. do exactly what you would expect them to do if you are familiar with Common Lisp.

Now we create a main program that populates the data base and prints a report on a family.

```
////////////////////////////////////////////////////////////
// Create and test a DataBase.

main() {
  DataBase db = DataBase();

  db.addEntity(S(smith-family));
  db.setValue(S(smith-family), S(people),
              list(S(joe), S(frank), S(mary)));

  db.addEntity(S(joe));
  db.setValue(S(joe), S(name), L("Joseph Clyde Smith"));
  db.setValue(S(joe), S(age), L(42));
  db.setValue(S(joe), S(siblings), list(S(frank), S(mary)));

  db.addEntity(S(frank));
  db.setValue(S(frank), S(name), L("Frank Bob Smith"));
  db.setValue(S(frank), S(age), L(32));
  db.setValue(S(frank), S(siblings), list(S(joe), S(mary)));

  db.addEntity(S(mary));
  db.setValue(S(mary), S(name), L("Mary Ann Smith"));
  db.setValue(S(mary), S(age), L(28));
  db.setValue(S(mary), S(brothers), list(S(joe), S(frank)));

  cout << "--- Report on Smith family ---" << endl;
  dolist(person, db.getValue(S(smith-family), S(people))) {
    cout << "Person: " << db.getValue(person, S(name)) << endl
         << "is " << db.getValue(person, S(age))
         << " years old" << endl;
    dolist(relation, list(S(siblings), S(brothers))) {
      let relatives = db.getValue(person, relation);
      if (relatives) {
        cout << "With " << relation << ": ";
        dolist(relative, relatives) cout << " " << relative;
        cout << endl;}}
    cout << endl;}}
```

Notice there are several occurrences of `S()` such as `S(joe)`. This is the Lpp idiom for introducing symbols. Symbols are useful since they are very efficient and have universal identity. For example `S(frank)` refers to the same symbol `frank` every place that it is used in any program.

Also notice the occurrences of `L()`. This is the Lpp idiom for introducing C++ primitive types into the Lpp world of objects. `L` stands for Lpp conversion because in essence we are

converting a C++ primitive value into an Lpp one. For example `L(42)` generates an Lpp Integer whose value is 42.

Finally notice that Lpp objects can be input and output to standard C++ streams. For example

```
cout << " " << relative;
```

outputs the value of `relative` to the C++ standard output stream `cout`. The variable `relative` is dynamically bound to an Lpp object in the `dolist` macro.

Compiling the whole data base program above and running produces the following output:

```
--- Report on Smith family ---
Person: Joseph Clyde Smith
is 42 years old
With siblings:  frank mary

Person: Frank Bob Smith
is 32 years old
With siblings:  joe mary

Person: Mary Ann Smith
is 28 years old
With brothers:  joe frank
```

1.5 Pure Lisp Include File

If you do not plan to use any C++ stream I/0 operators or C++ stream type arguments in any given C++ file then you can use

```
#include <LppPure.hh>
```

instead if `Lpp.hh`.

This will result in slightly faster compiles and slightly smaller binaries. `LppPure.hh` is the Pure Lisp include file and `Lpp.hh` includes it. `LppPure.hh` does not include any files, it is totally self contained. If in a given compilation unit you are using no I/O or are only using Lisp style I/O such as `read`, `print`, `prin1`, ... etc. then `LppPure.hh` is sufficient. The Lpp library itself for example uses `LppPure.hh` in most of its compilation units.

In this document whenever we talk about setting up optional compilation parameters before including the `Lpp.hh` file this also applies to wherever you are only including `LppPure.hh`.

If none of this matters to you, you can just use `Lpp.hh` all the time, it is always perfectly safe.

2 Name Space Issues

2.1 Naming Conventions

The Lpp library tries to prevent name space clashes with other libraries so that it can be included with such other libraries. All the identifiers reserved for Lpp fall into one of the four following categories

1. The Lpp class and Lpp subtype class names
2. Lpp member function identifiers
3. Lpp macros
4. The rest of Lpp reserved identifiers that begin with `Lpp_`

In case **1** Lpp class and subtype class names are few and correspond directly with names in the Common Lisp type hierarchy except that the Lpp class names begin with a capital letter. For example: `Number` for the Lpp number object, `Integer` for the Lpp Integer object, `String` for the Lpp string object, etc.

In case **2** the Lpp library creates its own name space of functions that dispatch on the `let` type. For example `list(x, y)` where `x` and `y` are of type `let`. Generally this prevents clashes with other library names.

Lpp strives to provide the Common Lisp specified semantics and wherever possible functions are named for their Common Lisp counterparts and replacing dashes by capitalizing the first character after the dash. For example the Common Lisp function `make-string` becomes `makeString`. This was decided over replacing dashes with underscores since such identifiers with capitals would more likely be unique and, except for the prefix `Lpp_`, allows a whole set of user defined identifiers with underscores that can not possibly clash with Lpp identifiers. Other characters not allowed in C++ identifiers are replaced by words, so `string=` becomes `stringEqual`. In some cases this does not work in which case a reasonable alternative is chosen. For example `append` is an identifier used in the C++ stream operations. So the identifier `listConcat` was used instead of the Common Lisp specified `append`. Another example is the Common Lisp specified `delete` which is a reserved identifier in C++, so `nremove` is used as in "destructive remove".

In case **3** the Lpp macros reserve a small set of reserved identifiers. A mechanism is set up to redefine these per compilation unit by the programmer if a name clash happens to exist, See Section 2.3 [Redefining Predefined Names], page 12.

In case **4** all other reserved Lpp identifiers begin with `Lpp_` so the user should never define a name that begins with `Lpp_`.

2.2 Returning C++ type names

In some cases in Lpp where it might be useful and more efficient to return a primitive C++ type, such as an `int`, the Common Lisp borrowed function name is used, since it is expected in C++ programs that that name will be used more frequently. However a counterpart function is still needed to return the corresponding Lpp object, such as an Lpp Integer object. Such counterpart functions can also be coerced to a function object for use in `funcall` and `apply`. In these cases if the Lpp function that returns the primitive C++ type is named

function

the the counterpart that returns the corresponding Lpp type is named

*function*L

The capital L is meant to connote that an Lpp object is the result. For example the Common Lisp borrowed name `length` that returns the length of a sequence is used in Lpp to return an `int` instead of an Lpp Integer object

```
int i = length(list);
```

and the counterpart function name is `lengthL`

```
let i = lengthL(list);
let i = funcall(L(lengthL), list);
```

The `length` function is more efficient and does not generate an Lpp Integer object. Whereas the `lengthL` does generate an Lpp Integer object if needed and as seen above can also be used as a function object whereas the primitive `length` can not, See Chapter 4 [Functions], page 23.

This is an unfortunate situation, since it would be better to have the function, such as `length`, be overloaded on both the `int` returning function and the `let` returning function. But some C++ compilers issue an ambiguous function name error in this case.

For efficiency concerns this is consistently done in all cases where the Common Lisp borrowed name of a function returns an integer. In such cases where the Common Lisp function would return either a integer or `nil` then a `-1` is returned for the `int` returning version. For example the Common Lisp function `list-length` returns the length of a given list as an interger but returns `nil` if the list is circular

```
int i = listLength(list);        // i == -1 if list is circular
let i = listLengthL(list);       // i == nil if list is circular
```

We mention the unfortunate situation above, but from another perspective the *function*L function name comments the fact that an Lpp object will be generated on the heap. To illustrate this, following is bad

```
if (listLengthL(list)) // then do something
```

Although it would work, the `listLengthL` returned value is not referenced anywhere and hence would cause a memory leak. It would be better done with

```
let len = listLengthL(list);
if (len) // then do something
```

or if the Lpp Integer object is not needed

```
if (listLength(list) > -1) // then do something
```

2.3 Redefining Predefined Names

Lpp defines a few identifiers that are only one or two letters for ease of typing since they are so frequently used in Lpp programs. By using capital letters as the first or second letter of these identifiers it is unlikely that there will be identifier clashes, but in the event that there are clashes Lpp allows all of these to be easily redefined. It is not recommended to redefine these if there are no clashes but nothing will hurt except non-consistency between Lpp programs.

The following are reserved identifiers that can be redefined per compilation unit:

```
True Nil O S L cL iL pL sL EM
```

Assuming that `XX` is the user's preferred redefinition, each can be renamed by putting one set of the following definitions in the compilation stream before `Lpp.hh` is included:

```
// Renames True
#define LPP_True_NODEFINE 1
#define XX Lpp_True

// Renames Nil
#define LPP_Nil_NODEFINE 1
#define XX Lpp_Nil

// Renames O
#define LPP_O_NODEFINE 1
#define XX Lpp_O

// Renames S
#define LPP_S_NODEFINE 1
#define XX Lpp_S

// Renames L
#define LPP_L_NODEFINE 1
#define XX Lpp_L

// Renames cL
#define LPP_cL_NODEFINE 1
#define XX Lpp_cL

// Renames iL
#define LPP_iL_NODEFINE 1
#define XX Lpp_iL

// Renames pL
#define LPP_pL_NODEFINE 1
#define XX Lpp_pL

// Renames sL
#define LPP_sL_NODEFINE 1
#define XX Lpp_sL

// Renames EM
#define LPP_EM_NODEFINE 1
#define XX Lpp_EM
```

3 Subtypes

Lpp achieves dynamic typing capability by having all Lpp objects be of type `let`. The type `let` is actually an `Let*` where `Let` refers to the base class for all specific Lpp types like Symbol, Cons, Integer etc. The actual base class of Lpp objects is abstracted away by the `let` and `Let` type definitions.

Thus while working with Lpp objects all objects are considered to be one cell pointers. However since the programmer never uses `Lpp*` or `Lpp` the concept of pointers is abstracted away with the use of `let`. So it is also correct to refer to Lpp objects as `let` objects (objects of type `let`). Furthermore all computational aspects of pointers is encapsulated in the Lpp classes. This means that in essence the programmer never has to deal with pointers while working specifically with `let` objects. This is consistent with the way Lisp works. In addition to this the programmer can declare `let` members in conventional C++ classes where non-abstracted pointer representations are needed.

The Lpp class defines a slot that contains a pointer to a type meta-object of type `Type`. This pointer is a `let` type and so the type meta-object itself is also a an Lpp object. `Type` objects themselves being of base type `let`, can be stored and operated on like any other Lpp object. So for example

```
let x = S(foo);
let y = list(x, typeOf(x));
cout << first(y) << second(y) << endl;
```

would print `x` followed by its type.

`Type` objects contain meta information about the specific objects it is a type of. Some of the information it contains are a type name, a type lattice member, and type *dispatching functions*. Type dispatching functions are very similar to virtual functions, but are more powerful in that they themselves can be dynamically manipulated. An example of type dispatching functions are the various printing functions for Lpp objects.

3.1 Predefined Subtypes

The predefined subtypes in Lpp are:

- **Type** Type meta-objects mentioned above.
- **Function** Analogous to Lisp first class function objects.
- **Cons** Analogous to Lisp cons cells.
- **Symbol** Analogous to Lisp symbols.
- **Number** Analogous to Lisp numbers.
- **Integer** Analogous to Lisp true integers.
- **Character** Analogous to Lisp characters.
- **String** Analogous to Lisp strings.
- **HashTable** Analogous to Lisp hash tables.

In addition to these Lpp subtypes the user can define his own Lpp subtypes which can then be manipulated as `let` objects. This is done with the `classL` macro that hides away the details of declaring such a class, See Section 3.3 [Defining New Subtypes], page 16.

3.2 Conversions

Conversions from primitive C types to Lpp objects is done with the *L* constructor and back to primitive C types is done with the *xL* converter, where *x* designates a primitive C type.

Implicit constructors were avoided in Lpp for good reasons. One major reason is that it eliminated overloaded name conflicts with other C++ libraries. Another reason is that 0 as `nil` and 0 as integer is problematical. This may seem like a real inelegancy. For example the programmer has to specify `list(L(55), L("string"), L(fun))` instead of simply `list(55, "string", fun)`, but as it turns out this is not much of a problem in actual programming practice. Also, provided functions like `listSEM("one", "two", "three", EM)` for long homogeneous lists of like objects helps.

The name of the `L` constructor can be redefined per compilation unit, See Section 2.3 [Redefining Predefined Names], page 12.

3.2.1 From C++ Primitive Types

In Lpp the *L* macro is used for converting any basic C type to an Lpp object.

`L` *x* [Macro]

This macro returns an Lpp object that is analogous to the type of the argument *x*. For example if *x* is a `int` then an Lpp Integer object is returned. Note that no part of the returned Lpp object shares the value of *x*. Some actual examples are:

```
// Converting a ...
let myInteger = L(5);              // C integer to Lpp Integer
int x = 123;
let myInteger2 = L(x)              // C integer to Lpp Integer
let myCharacter = L('z')           // C char to Lpp Character
let myString = L("Some string");   // C string to Lpp string
let myFun = L(cFunction);          // C function to Lpp function
```

The Lpp object types created by the `L` conversion macro when applied to C++ types is as follows:

```
int => Lpp Integer
char => Lpp Character
char* => Lpp String
let fun(let, let ...) =>  Lpp Function
```

An Lpp Function object can be created from a function returning a type `let` taking 0 or more type `let` arguments.

3.2.2 To C++ Primitive Types

Converting an Lpp object back to primitive C types is done with the *xL* converter, where *x* designates a primitive C type.

`iL` *x* [Function]

`cL` *x* [Function]

`sL` *x* [Function]

`pL` *x* [Macro]

The function `iL` converts an Lpp Integer *x* to and `int` and returns, `cL` converts an Lpp Character *x* to a `char` and returns, `sL` converts an Lpp String *x* to a `char*` string and returns, `pL` converts an Lpp predicate (nil or t) to a C predicate (0 or 1). Some examples are:

```
// Converting a ...
let myInteger = L(5);              // C integer to Lpp Integer
int i1 = iL(myInteger);            // Then back to int
let c1 = L('z' );                  // C char to Lpp character
char ch = cL(c1 );                 // Then back to C char
let myString = L("Some string");   // C string to Lpp string
char* s1 = sL(myString);           // Then back to a char*
```

3.3 Defining New Subtypes

The user of Lpp can create new Lpp subtypes with the `classL` and `classLS` macros. Note that all examples given in this section assume that a type meta-object is generated somewhere (see `genT` and `genTS` macros below). When a class is defined as an Lpp subtype a new type meta-object named by that class name will exist in the Lpp type hierarchy. Instances of that class then can be used anywhere an Lpp object of type `let` can be used. For example, such an instance can then occur in an Lpp list.

`classL` *name* [Macro]

`classLS` *name* [Macro]

The `classL` macro is used for defining new Lpp subtypes of type `let`. The `classLS` macro is used for defining new Lpp subtypes of Lpp supertypes. The *name* argument specifies the name of the new class and subtype. The syntax of both is exactly the same as the ordinary C++ `class` syntax except that instead of using the `class` identifier either `classL` or `classLS` is used.

For example

```
classL(MyType) {...};
let m1 = makeInstance(MyType);
let list1 = list(L(1), m1, L(2));
cout << second(list1);
```

We could do a similar thing with a subtype of `MyType`:

```
classLS(MySubType) : private MyType {...};
let ms1 = makeInstance(MySubType);
let list1 = list(L(1), ms1, L(2));
cout << second(list1);

// Should print "t" for both of these:
cout << typep(second(list1), type(MyType));
cout << typep(second(list1), type(MySubType));
```

To pass the specific type of any new Lpp object down through the Lpp type lattice any constructor in a `classL` or `classLS` definition *must* pass in a constructor for the super type with the subtype as an argument. For `classL` this would be the `Let` supertype and for

`classLS` the user defined Lpp type. Recall that the Lpp type can be referenced using the `type` macro. So for a new user defined Lpp type `T` the super constructor needed for `classL` would be `Let(type(T))`. Or using our example above for `MyType` and `MySubType` the class constructors might appear as follows

```
MyType() : Let(type(MyType)) {...}
MySubType() : MyType(type(MySubType)) {...}
```

`makeInstance` *type* [Macro]

Returns a new Lpp object of type *type*. The argument *type* can be a C++ constructor form or just the type name. The constructor or type name must be of an Lpp type.

Since all Lpp object subtypes must have a type meta-object, See Section 3.4 [Accessing Type Meta-Objects], page 17, the Type object must be generated somewhere, usually in the C++ file that defines the accessors for the subtype. This can be done easily with the `genT` or `genTS` macros.

`genT` *type* [Macro]

`genTS` *type supertype* [Macro]

The `genT` macro generates a type meta-object of type *type*. The `genTS` macro generates a type meta-object of type *type* whose supertype is *supertype*. For example

```
classL(MyType) {...};

// The type meta-object for a MyType is generated here
genT(MyType);

classLS(MySubType) : private MyType {...};

// The type meta-object for a MySubType is generated here
genTS(MySubType, MyType);
```

3.4 Accessing Type Meta-Objects

As mentioned previously all Lpp objects are of some type in the Lpp type hierarchy where each type has a `Type` meta-object associated with it. We call these *meta* objects because they contain the information about and type functionality of the ordinary Lpp objects they are associated with. There is one type meta-object for all ordinary Lpp object instances of that type.

These `Type` objects are first class Lpp objects and can be manipulated and accessed dynamically. For example the print methods of specific types can be accessed and set dynamically. First we show two functions for getting the type meta-object itself.

`typeOf` *object* [Function]

This function returns returns the type meta-object for the type of the Lpp object *object*.

`type` *name* [Macro]

This macro returns the type meta-object named *name*. Here are some examples:

```
let typeList =
  list(type(Integer), type(Symbol), type(MyType));
eq(typeOf(L(23)), first(typeList)) => t
eq(typeOf(S(23)), second(typeList)) => t
```

`typeName` *object* [Function]

`typeNameL` *object* [Function]

`typeName` returns a `char*` representing the type print name of the object *object*. For example:

```
let object = S(red);
typeName(object) => "Symbol"
```

The `typeNameL` function does the same but returns an Lpp String object.

A `Type` meta-object contains various functions that dispatch on the type of a given object dynamically. These are similar to virtual functions but are more powerful in that they can be manipulated dynamically. Lpp provides some built in type dispatching functions for things such as printing and equality of objects.

All Lpp `Type` meta-objects when created have default dispatching functions generated. The user however may access these dispatching functions or set them to new dispatching functions dynamically. In general there are slots in the `Type` meta-object that can be accessed by functions of the form

```
setTypeXXXX type data
getTypeXXXX type
```

where `XXXX` refers to the kind of data being accessed in the given type where `type` is an Lpp `Type` meta-object. And `setTypeXXXX` sets the `data` and `getTypeXXXX` returns the data `data` last set. When setting `data` is given as `nil` it usually gets interpreted as setting back to the a default. This setting and getting can be done dynamically at run time. Here are some examples

```
let original = getTypePrinc(type(MyType));
setTypePrinc(type(MyType), L(MyTypePrincFunction));
setTypePrinc(type(MyType), Nil);
setTypePrinc(type(MyType), original);
```

The first line sets `original` to the current `princ` printer function of `MyType`. The second line sets the `princ` printer function of `MyType` to `MyTypePrincFunction`. The third line would set the `princ` printer function to the Lpp default. And the last line sets the `princ` printer function back to what it was originally.

`setTypePrinc` *type function* [Function]

`getTypePrinc` *type* [Function]

`setTypePrin1` *type function* [Function]

`getTypePrin1` *type* [Function]

All Lpp objects have `princ` and `prin1` methods used by the Lpp printing family of functions and C++ stream IO of Lpp objects, See Chapter 14 [Input Output], page 57. These functions allow the user to set or get the print method of a given type where *type* is a `Type` meta-object. The functions `setTypePrinc` and `setTypePrin1` sets the Lpp function

object for the `princ` and `prin1` methods respectively. The *function* argument is a function taking two arguments, first an object of type `let` and second a stream of type `ostream&`. The *function* should return the first argument object as both `princ` and `prin1` do. The functions `getTypePrinc` and `getTypePrin1` returns the last function set.

For example suppose that we had defined an Lpp class called `MyClass` that has two slots with accessors `getSlot1` and `getSlot2`. Then the following code would set up a `princ` type printer for `MyClass`

```
let princMyClass(let obj, ostream& s) {
  s << "MyClass object: slot1: "
    << getSlot1(obj) << "  slot2: " << getSlot2(obj);
  return obj;}

let mc1 = makeInstance(MyClass(76, "trombones"));

cout << mc1 << endl; // Would produce

        <Lpp Testclass>

setTypePrinc(type(MyClass), L(princMyClass));  // Then

cout << mc1 << endl; // Would produce instead

        MyClass object: slot1: 76  slot2: trombones
```

The default `princ` and `prin1` type dispatching functions for any Lpp type will print an object as

```
<Lpp xxxx>
```

where `xxxx` will be the type name of the object. The `princ` type dispatching function is the one used by the `princ` function and also by the `<<` operator as seen above.

`setTypePrins` *type function* [Function]

`setTypePrins` sets both the `princ` and `prn1` printing function to the same *function*.

When an Lpp equality predicate of either `equal` or `equalp` is called, an equality dispatcher function of the `Type` meta-object is called to compute the predicate. The two arguments of the equality predicate are passed to the dispatcher function and the returned result is the result of the equality predicate. The equality dispatcher functions can be accessed with the following

`setTypeEqual` *type function* [Function]

`getTypeEqual` *type* [Function]

`setTypeEqualp` *type function* [Function]

`getTypeEqualp` *type* [Function]

The `setTypeEqual` function sets the equal dispatcher function of the `Type` meta-object *type* to the given function *function* which is an Lpp function object of two arguments. When `equal` is called given two objects where the first object is of this type then the function *function* is automatically dispatched on the object. Given both objects the function should

return `t` if the objects are considered to be "equal" and `nil` otherwise. As an example suppose that we had defined an Lpp class called `MyClass` then

```
let myClassEqual(let o1, let o2) {
      return equal(the(MyClass, o1).slot1, the(MyClass, o2).slot1);}

let mc1 = makeInstance(Myclass("foo"));
let mc2 = makeInstance(Myclass("foo"));

equal(mc1, mc2) => nil
setTypeEqual(type(Myclass), L(myClassEqual));
equal(mc1, mc2) => t
```

`setTypeEqualp` does the same thing as `setTypeEqual` except that when `equalp` is called given two objects where the first object is of the type *type* then the function *function* is automatically dispatched on the object. Given both objects the function should return `t` if the objects are considered to be "equalp" and `nil` otherwise. `getTypeEqual` and `getTypeEqualp` will return the last `equal`, `equalp` dispatcher functions respectively set for the given *type*.

If a type meta-object does not have a equality dispatcher function set this way then `eq` will be used as the default.

`setTypeExt` *type extension* [Function]

`getTypeExt` *type* [Function]

The Lpp user can set up his own Type meta-object extension to any Lpp Type meta-object. He first defines an Lpp class that will be the meta-object extension class and then instantiates a object of that class which he then sets in the desired Type meta-object by calling `setTypeExt` on the desired Type *type* and meta-object *extension* is that newly instantiated object. `getTypeExt` would return the last extension set in the given *type* meta-object.

As an example this is useful when over some number of classes the user wants to dispatch on a common action, like the `princ` printing action above. Since the extension is a first class Lpp object it can be exchanged dynamically. So, for example, the user can cause a collection of classes to behave one way in one mode and then another way in a different mode. Also all Lpp objects Type meta-objects are guaranteed to have an extension even if only `nil` which would usually be a default case or no action.

3.5 Accessing Subtypes

In the majority of code written using Lpp, Lpp objects appear as type `let`. All operations within Lpp objects are done with accessors. When defining new Lpp objects the user needs to get at the internals of such objects for defining his own accessors. The Lpp `the` macro makes it easy and safe to access the internals of an Lpp object.

`the` *name exp* [Macro]

`theOrNil` *name exp* [Macro]

`asThe` *name exp* [Macro]

`fromThe` *name exp* [Macro]

The argument *name* names some subtype of the Lpp type hierarchy. Lpp takes the liberty to adapt the Common Lisp *the* construct. Much like Common Lisp the Lpp `the` macro evaluates the given expression *exp* and returns the value if it is of the type named by the given *name* and triggers an exception if the type is not of the given *name*. In addition the Lpp `the` macro returns a pointer to that object of type name *name*. This is a specific pointer of that *name* type as opposed to the more general Lpp `let` type. For example:

```
classL(MyType) {
    int slot1;
    public: friend void setSlot1(let, int);};
setSlot1(makeInstance(MyType), 25);

void setSlot1(let x, int val) {    // x is input as general let type
    MyType* mt = the(MyType, x);   // mt is now a specific pointer
    mt->slot1 = val;}              // to a MyType object
```

The `the` macro enforces that the given object is of type *type* by doing a dynamic type check. Dynamic type checking can be turned off per compilation unit, See Section 3.6 [Type Checking], page 21.

The `asThe` macro is the same as `the` but never does any type checking. It is useful when a function absolutely knows what the type of an Lpp object is. The `fromThe` macro is the same as `asThe` but instead demands that the object is of the given type or derived from a supertype of the given type. It is useful when a function absolutely knows what the type or supertype of an Lpp object is. The `theOrNil` macro is the same as `the` but also allows `nil` to pass as the type.

3.6 Type Checking

When dynamically typed objects are being processed in Lpp in a type safe mode, objects are dynamically checked for the expected type. In Lpp this is enfored with the *the* macro, See Section 3.5 [Accessing Subtypes], page 20. This usually only occurs at one place in object accessors. When it is not of the expected type control is transferred to a specified handler, usually to an error code handler.

The default is to have this dynamic type checking turned on. It can be turned off per compilation unit with the following `define` that must precede the include of `Lpp.hh`

```
#define LPP_NO_TYPE_CHECK 1
```

This automatically modifies the behavior of the `the` macro. With type checking turned off the users code using Lpp is just as efficient as ordinary C++ code accessing statically compiler type checked references. Rarely do you want to turn off type checking while developing a program and usually doesn't even hurt the efficienty too much on a delivered package.

The above methods turns off type checking per your compilation units that see `LPP_NO_TYPE_CHECK` as defined above. If you also want the Lpp library itself to not do type checking then you need to recompile the Lpp library in the same way as above or better yet link your program with the Lpp library where this has already been done as for example linking with the Lpp supplied library

```
libLppNC.a
```

instead of `libLpp.a`. You can optionally make and install `libLppNC.a` in the Lpp installation, See Section 1.2 [Installing Lpp], page 4.

4 Functions

As with Common Lisp Lpp has first class function objects. That is to say that objects can be created that represent C++ functions and manipulated like any other Lpp object. Such function objects are easily created with the `L` conversion macro. For example

```
let someFun(let x, let y) {...}
let functionList = list(L(car), L(someFun))
```

In this example `functionList` contains a list of two function objects: `car` an existing Lpp function and some other function object called `someFun`. In general any function object can be generated from any C++ function with an address (non-inline) of type `let` with 0 or more `let` type arguments. Note that Lpp functions that return `int`s can not be used but they have counterparts that can. For example `length` can not but `lengthL` can.

Such function objects can be called using either `funcall` or `apply`, See Chapter 5 [Control Structure], page 24. For example if in the previous example `someFun` given two arguments returns some list then

```
let test(let a, let b, let functionList) {
  return funcall(nth(0, functionList),
                 funcall(nth(1, functionList), a, b));}
```

would return the car of that list. Actually notice that `test` in this example could be used to apply any Lpp function of one argument to any Lpp object returned by any function of two arguments. Using first class function objects as variables adds an important dimension to programming.

`ARGSn` [Macro]

Where `n` is the number of function arguments. For example: ARGS0, ARGS1, ARGS2 ... etc. This macro makes it easy to cast function objects of functions that can have more than one argument such as `list`. For example:

```
let fun = L(ARGS3 list);
print(fun) => <Lpp Function of 3 args>
funcall(fun, S(a), S(b), S(c)) => (a b c)
```

Lpp only supplies up to 10 arguments for such functions but a quick look at the Lpp include file `LppPure.hh` makes it apparent that any number of arguments can be added easily. However, very rarely are more than 10 arguments needed and if you are habitually using more than 10 arguments then you are probably doing something wrong or else you should be using something like `listEM` or `listSEM`, See Section 11.2 [List operations], page 50.

5 Control Structure

Lpp tries as much as possible to capitalize on C++ control structure. C++ control structure is augmented however with the use of Lpp first class function objects and functions `funcall` and `apply`. Lpp supplies control structure macros to make code easier to read and implement. For example the `dolist` macro just expands into a harder to read C++ `for` loop.

5.1 Function Invocation

As mentioned previously, See Chapter 4 [Functions], page 23, Lpp first class function objects can be generated and manipulated as data and also applied to other data.

`funcall` *fun args* [Function]

This function calls the Lpp function object *fun* on the *args* which can be 0 or more arguments of type `let`. A value of type `let` that *fun* returns is returned from the `funcall`.

`apply` *fun args* [Function]

This function is similar to `funcall` except that the arguments given to *fun* is taken from only one argument *args* which is a list of 0 or more Lpp objects.

Here are some examples that contrast `funcall` with `apply`:

```
let list1 = list(L("Lost"),  L("World"));
let fun = L(listConcat);
funcall(fun, cadr(list1), car(list1)) => WorldLost
apply(fun, list1) => LostWorld
```

5.2 Iteration

5.2.1 Iteration Constructs

`dolist` *el list* [Macro]

This macro loops through the Lpp list *list* with a given variable name *el* bound to the elements of the list. The variable name *el* will be bound to the last element of the list on exit from the loop or else `nil` if the list *list* was empty. For example the following function would print the given `list1` entries in a single column

```
void printListColumn(list1) {
  dolist(e, list1) cout << e << endl;}
```

`dolist2` *el1 list1 el2 list2* [Macro]

This macro loops through the Lpp lists *list1* and *list2* with variable names *el1* and *el2* bound to the elements of the two lists. The variable names *el1* and *el2* will be bound to the last element processed from the list on exit from the loop or else `nil` if the list *list* was empty. If the lists are not the same length the shorter of the two lists terminates the loop.

5.2.2 Mapping

The list mapping functions all take a function argument *fun* as the first argument then 1 or 2 list arguments *lists*. The function argument *fun* must have as many arguments as there

are list arguments. The given list or lists are mapped over or into a new list using *fun*. The function *fun* is successively applied to the list or lists and its return value, except for `mapc` and `mapl`, is used to construct a new list in the same order as the given `lists`. For `mapc` and `mapl` *fun* is successively applied for side effect only and the original first list is just returned. For `mapcar`, `maplist`, `mapcan` and `mapcon` the newly mapped list is returned.

If more that one list is given then the shortest list terminates the mapping.

`mapcar` *fun lists* [Function]

`maplist` *fun lists* [Function]

The lists *lists* are mapped over by *fun* and the mapped list is returned. `mapcar` maps over successive elements of the lists which are passed to *fun*. `maplist` maps over successive `cdr`s of the lists which are passed to *fun*. For example:

```
let list1 = list(L("one"), L("two"), L("three"));
mapcar(L(lengthL), list1) => (3, 3, 5)
let list2 =
  list(L(" cat"), L(" dogs"), L(" bears"), L(" lions"));
mapcar(L(stringConcat), list1, list2)
        => ("one cat", "two dogs", "three bears")
maplist(L(lengthL), list1) => (3, 2, 1)
maplist(L(identity), list1)
        => (("one" "two" "three") ("two" "three") ("three"))
```

`mapc` *fun lists* [Function]

`mapl` *fun lists* [Function]

The lists *lists* are mapped over by *fun* and the first list, ie. the second argument, is returned. `mapc` maps over successive elements of the lists which are passed to *fun*. `mapl` maps over successive `cdr`s of the lists which are passed to *fun*. For example:

```
let list1 = list(L("One"), L("Two"), L("Three"));
let (*princOne)(let) = &princ;
mapc(L(princOne),list1);  // Prints on cout: OneTwoThree
```

The mapping functions `mapcan` and `mapcon` are the same as `mapcar` and `maplist` respectively except that it is as if `nconc` were applied to the resulting values returned by *fun*. It is important to notice that because of this the values returned by *fun* are concatenated together by destructively modifying them. Usually the function *fun* creates and returns new lists each time being aware that they will be modified by being concatenated together.

`mapcan` *fun lists* [Function]

`mapcon` *fun lists* [Function]

The lists *lists* are mapped over by *fun* and the values returned by *fun* are concatenated together, as if by using `nconc`, and the concatenation is returned. `mapcan` maps over successive elements of the lists which are passed to *fun*. `mapcon` maps over successive `cdr`s of the lists which are passed to *fun*.

`mapcan` and `mapcon` can be used for combining or filtering. For example:

```
// Example of filtering:
let passNums(let x)
     {if (numberp(x)) return list(x); return Nil;}
let list1 = list(L(1), S(a), L(2), S(b), L(3), S(c));
mapcan(L(passNums), list1) => (1 2 3)
// Example of combining:
list1 = readFromString("((1 2 3) (4 5 6) (7 8 9 10))");
mapcan(L(copyList), list1)  => (1 2 3 4 5 6 7 8 9 10)
```

6 Predicates

An Lpp *predicate* is a function that returns either `nil` or a non-`nil` value based on a test of its given arguments. The return of a non-`nil` value implies *true* and the return of `nil` implies *false*. Things are done this way since a check for `nil` or non-nil is always efficient and uniform. Any other Lpp type `let` value other than `nil` is considered non-`nil`. This allows useful values to be returned that at the same time imply *true*. Furthermore having no specific object implying true makes multi value logics in programs easy and consistent with reductions to two valued logics.

Lpp uses the same philosophy as Common Lisp for deciding when to return non-`nil` or `t`. If no better non-`nil` value is available for indicating success, the symbol `t` will be returned.

6.1 Logical Values

Lpp chose to use 0 as `nil`. This allows Lpp code to be symmetric with usual C code that uses 0 and 1 as predicates. The alternative would be to have to use the `null` function for every predicate occurrence. For example any Lpp function `f` returning `let` or Lpp object `x` can be used as a predicate

```
if (x) // do something
if (f()) // do something
```

Note that `nil` serves as three important things, a logical value, the end of a list and it is also the symbol `nil`. The 0 as `nil` concept it is to be thought of as though the symbol object `nil` resides at memory address 0. So that everything that can be done with a symbol can be done with 0. For example

```
symbolName(0) => "nil"
```

When a `let` variable has the value 0 it prints as `nil`. Conversely when in any s-expression is read from a stream `nil` is translated into 0 internally. Lpp provides a global variable `True` that has as its value the Lpp Symbol `t`, See Chapter 7 [Symbols], page 31. And `Nil` is defined simply as 0, so it is just as efficient to use `Nil` as 0 in your code except that the compiler knows that `Nil` is an Lpp type.

Choosing 0 as `nil` creates an ambiguous situation when a type can be interpreted as either `int` or `let`. For example overloaded Lpp functions that are disambiguated between those that have the same function profile but differ in an `int` or `let` argument and so the following is legitimate

```
let someFunction(int i, let obj);  // i to be an int
let someFunction(let i, let obj);  // i to be an Lpp Integer or nil
```

So to disambiguate you would use `Nil` where `nil` was intended

```
someFunction(0, something);     // Means 0 as an int
someFunction(L(0), something);  // Means 0 as an Lpp Integer
someFunction(Nil, something);   // Menas 0 as Lpp nil
```

`Nil` [Constant]

`True` [Constant]

`Nil` is intended to be used in programs where the `nil` symbol is needed and `True` is intended to be used where the `t` symbol is needed. These are both names of constants

that can not be assigned other values. Since the Lpp `Nil` object prints as `nil` and the Lpp `True` object prints as `t`, `nil` and `t` when used in this manual will refer to those objects respectively.

The name of the constant `True` and `Nil` can be redefined per compilation unit, See Section 2.3 [Redefining Predefined Names], page 12. So for example an Lpp programmer could redefine these names to be `nil` and `t` to correspond to the Common Lisp constants `nil` and `t`. After examining many C++ programs it was decided that the default should not be the names `nil` and `t` since these frequently occurred as variable names. For example `t` is frequently used as a variable in C++ programs to denote "time". Also, using `Nil` and `True` in Lpp programs emphasizes the fact that they are just C++ identifier names and not in fact symbols as they are in Lisp programs.

`null` *exp* [Function]

`null` returns `t` if given expression `exp` evaluates to `nil` and returns `nil` otherwise.

6.2 V-tables and Type Predicates

Data type predicates determine the type of an Lpp object or compare type hierarchies. This is important for complex programs where objects must be dynamically typed and there are classes of algorithms that can not be easily implemented using v-tables ie. virtual function tables. The Lpp user can still use v-tables and mix the use of v-tables with dynamic type dispatching. Since these data type predicates are very efficient they should not be avoided and used wherever it seems reasonable. Furthermore Lpp provides for type dispatching functions in type meta-objects, See Section 3.4 [Accessing Type Meta-Objects], page 17, which are as efficient as v-tables but more flexible and dynamic.

6.3 General Type Predicates

`typeIs` *exp name* [Macro]

`typeIs` returns `t` if the expression *exp* evaluates to an object whose type is exactly of the type name *name* and `nil` otherwise. Note that "exactly" implies that if *name* is a supertype of the object's type then false is returned. This macro is designed to be as fast a type check as possible and thus returns a low level C predicate, 0 for false and 1 for true. The returned result can easily be converted to the Lpp predicate with the `pL` macro if needed.

`typep` *object type* [Function]

`typep` returns `t` if the object *object* type or any supertype is the `Type` meta-object *type* and `nil` otherwise. Note that `typep` is different from `typeIs` in that any supertype of the object satisfies the test. Here are some examples of both.

```
let n = L(56);
pL(typeIs(n, Integer)) => t
pL(typeIs(n, Number)) => nil
typep(n, type(Number)) => t
```

6.4 Specific Data Type Predicates

`symbolp` *object* [Function]
`symbolp` returns `t` if *object* is of type `Symbol` and `nil` otherwise.

`consp` *object* [Function]
`consp` returns `t` if *object* is of type `Cons` and `nil` otherwise.

`listp` *object* [Function]
`listp` returns `t` if *object* is of type `Cons` or `nil` and `nil` otherwise.

`numberp` *object* [Function]
`numberp` returns `t` if *object* is of any type whose has a supertype of `Number` and `nil` otherwise.

`rationalp` *object* [Function]
`rationalp` returns `t` if *object* is of any type whose has a supertype of `Rational` and `nil` otherwise.

`integerp` *object* [Function]
`integerp` returns `t` if *object* is of type `Integer` and `nil` otherwise.

`ratiop` *object* [Function]
`ratiop` returns `t` if *object* is of type `Ratio` and `nil` otherwise.

`stringp` *object* [Function]
`stringp` returns `t` if *object* is of type `String` and `nil` otherwise.

`characterp` *object* [Function]
`stringp` returns `t` if *object* is of type `Character` and `nil` otherwise.

`functionp` *object* [Function]
`functionp` returns `t` if *object* is of type `Function` and `nil` otherwise.

`hashTableP` *object* [Function]
`hashTableP` returns `t` if *object* is of type `Hashtable` and `nil` otherwise.

6.5 Equality Predicates

Lisp has taught us that there are several kinds of equality of objects. In C++ the `==` operator can only express one kind of equality. Also without `void*` casting it can not be applied to any two objects dynamically.

The `==` operator in Lpp can be applied to any two Lpp objects dynamically and means the same as the `eq` function of Common Lisp. However an `eq` function is still supplied so that an `eq` function object can be passed to some functions that take an optional predicate function argument. Also note that `eq` as opposed to `==` is a true Lpp predicate in that it returns either `t` or `nil`.

`eq` *x y* [Function]
`eq` returns `t` if *x* is the same object as *y* and `nil` otherwise.

`eql` *x y* [Function]

`eql` is the same as `eq` except that if *x* and *y* are `Character`s or `Number`s of the same type their values are compared.

`equal` *x y* [Function]

`equal` returns `t` if *x* and *y* are structurally similar (isomorphic) objects and `nil` otherwise. Roughly speaking, `t` means that *x* and *y* print the same way.

`equalp` *x y* [Function]

`equalp` returns `t` if *x* and *y* are are `equal`; if they are characters and are `eql` ignoring alphabetic case; if they are numbers and have the same numerical value, even if they are of different types; or if they have components that are all `equalp`.

Objects that have components are `equalp` if they are of the same type and corresponding components are `equalp`. This test is implemented in a recursive manner and may fail to terminate for circular structures. For conses, `equalp` is defined recursively as the two car's being `equalp` and the two cdr's being `equalp`.

Here are some example of all these equality predicates

```
eq(L(23), L(23)) => nil
eql(L(23), L(23)) => t
equal(L(23), L(23)) => t
eq(L('a'), L('a')) => t
eql(L('a'), L('a')) => t
equal(L('a'), L('A')) => nil
equalp(L('a'), L('A')) => t
eq(S(foo), S(foo)) => t
eql(S(foo), S(foo)) => t
equal(S(foo), S(foo)) => t
equal(S(foo), S(Foo)) => nil
equalp(S(foo), S(Foo)) => nil
eql(L("foo"), L("foo")) => nil
equal(L("foo"), L("foo")) => t
equal(L("foo"), L("Foo")) => nil
equalp(L("foo"), L("Foo")) => t
equalp(S(foo), L("foo")) => nil
```

7 Symbols

Symbols are Lpp objects of type `Symbol` that have universal identity and usually stand for what they connote. For example in a program the symbol `red` will be the same symbol as `red` in another program and would most likely connote the color red. C++ `enums` do not have this property of universal identity. Furthermore symbols are a lot more efficient than strings for this purpose since there is only one symbol object in a system per symbol name such as `red`. When a symbol is introduced into the system it is said to be *interned*.

7.1 The Print Name

A useful property of symbols is that they print the way they read. In most program code the name of a symbol, called the *print name*, is rarely needed, usually only the `let` type object is used as the identity of the Symbol object. When the name of the symbol is needed, such as for printing purposes or text manipulation the following function is used.

`symbolName` *symbol* [Function]

`symbolName` is given an Lpp Symbol object *symbol* and the print name is returned as an Lpp String, See Chapter 13 [Strings], page 56. For example:

```
let sym = S(red);
length(symbolName(sym)) => 3
```

`symbolName` does not generate any new Lpp objects since the string object of a symbol is only generated once and used over and over again. It is for this reason that even thought the name string object of a symbol is accessible that string should not be modified from its original since it would render that symbol unusable.

7.2 Creating Symbols

A symbol can be introduced into the system with the `intern` function.

`intern` *name* [Function]

`intern` returns the interned symbol with the name *name*. *name* can be either an Lpp String or a `char*` string. The *name* argument can be composed of any characters unless it is a `char*` in which case the character 0 is not allowed. If the symbol *name* has not been interned yet then it is interned first.

Lpp symbol are are case sensitive. So, for example, `Red` and `red` are two different symbols.

The `S` macro provides a short hand for entering symbols.

`S` *name* [Macro]

This macro returns an Lpp Symbol with print name *name*. The *name* argument is not escaped with double quotes. Note that for some sequence of characters *chars*

```
S(chars) == intern("chars")
```

For example:

```
let sym1 = S(red);
let sym2 = intern("red");
sym1 == sym2 => 1
eq(sym1, sym2) => t
```

Note that creating a symbol with `S` in an Lpp program is not the same as quoting a symbol in a Common Lisp program. In a Common Lisp program the quoted symbol causes `intern` to be called only on reading the program, however in a C++ program `intern` will be called when the code is running. So for example

```
let sym1 = S(red);
let list1 = list(sym1, sym1);
```

is slightly more efficient than

```
let list1 = list(S(red), S(red));
```

In both cases only one Symbol `red` is created, but in the second case `intern` is called twice, the second time simply returning the already interned symbol `red`.

`unintern` *symbol* [Function]

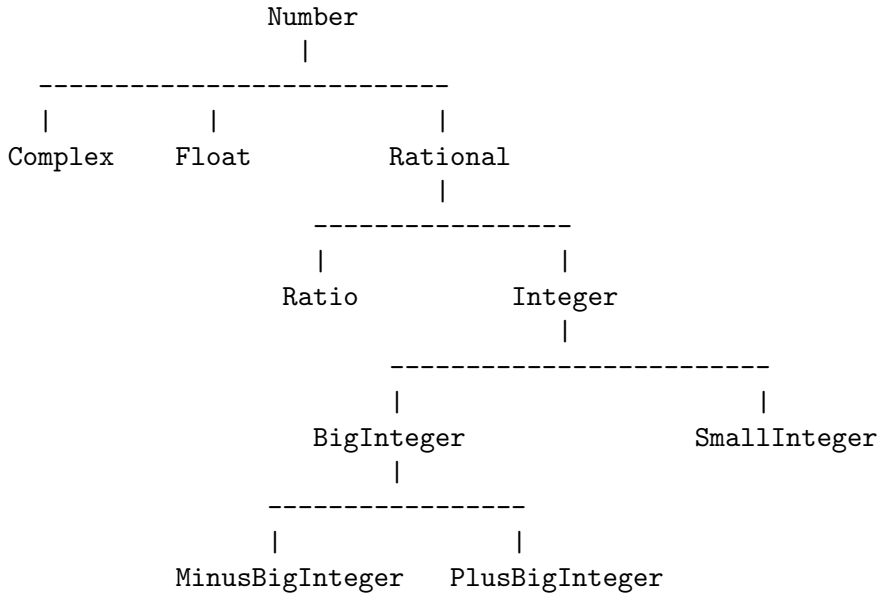
`unintern` removes the symbol *symbol* from the system. It returns nil.

8 Numbers

In Lpp all specific kinds of numbers, such as Integers, are subtypes of the Lpp type Number. Operations in general are provided for general Numbers as well as specific number types where necessary. If a function only makes sense for a number argument then an error will be signaled if that argument is not a number.

8.1 Number Types

Lpp's type hierarchy for numbers is depicted in the following diagram[1]



All of the number types fall into a more general type above them in the hierarchy. For example a `Rational` is also a `Number` type and an `Integer` is also a `Rational` type and a `Number` type, and so on and so forth. In Lpp this can be demonstrated by the code

```
let x = L(18);
typep(x, type(Number)) => t
typep(x, type(Integer)) => t
typep(x, type(SmallInteger)) => t
typep(x, type(BigInteger)) => nil
typep(x, type(Ratio)) => nil
```

The `Integer` type is by far the most important for doing symbolic mathematics where overflow or underflow can not be tolerated, such as systems that manipulate polynomials symbolically. Thus integers can fall into the range of minus infinity to plus infinity. Lpp implements this with the `BigInteger` type. When an integer starts out in Lpp it can be a `SmallInteger` or `BigInteger`

```
let x = L(21);
let y = readFromString("22222222222222222211111111111111111111);
```

[1] As of this writing the Complex and Float types have not been implemented.

In this example `x` starts off as a `SmallInteger` and `y` starts off as a `BigInteger`. More specifically `y` is a `PlusBigInteger`

```
print(typeOf(x)) => <Lpp Type for SmallInteger>
print(typeOf(y)) => <Lpp Type for PlusBigInteger>
```

As Lpp math functions are applied to an integer it can expand from a `SmallInteger` into a `BigInteger` and vice verse. If the following code were allowed to loop

```
while(1) {
  x = inc(x);
  y = dec(y);}
```

then `x` would eventually transform from a `SmallInteger` to a `BigInteger` and `y` from a `BigInteger` back into a `SmallInteger`. `BigInteger` objects can grow unlimited toward plus or minus infinity.

A `Rational` number can either be an `Integer` or a `Ratio`, where a `Ratio` is nothing more than a pair of `Integer` objects. The first of the pair being the numerator of the ratio and the second being the denominator. A `Ratio` number can start off being read from an s-expression stream or as the result of a math function

```
let a = readFromString("12/3");
let w = readFromString("21/222222222211111111111");'
let z = divide(x, y);
```

As with Common Lisp in Lpp any computation or notation that produces a ratio such that the numerator and denominator can be evenly divided by an integer then the numerator and denominator are immediately converted to the divided results. Or in other words the result is always immediately canonicalized. So given the variables `a` and `w` in the above example

```
print(a) => 4
print(w) => 7/7407407407037037037
```

Note that the canonicalization takes place immediately on the `readFromString` or `divide` above and not just on the `print`, so that there never exists a non-canonicalized rational number in Lpp.

8.2 Predicates on Numbers

These are predicate functions that test a condition of a single number argument and returns `t` if the condition tests true and returns `nil` otherwise.

`zerop` *x* [Function]
Tests for the condition that the given number *x* is zero.

`plusp` *x* [Function]
Tests for the condition that the given number *x* is non-zero and positive.

`minusp` *x* [Function]
Tests for the condition that the given number *x* is non-zero and negative.

`oddp` *x* [Function]
Tests for the condition that the given number *x* is odd.

`evenp` *x* [Function]

Tests for the condition that the given number *x* is even.

8.3 Comparison on Numbers

Numbers can be compared with the following functions.

`lessThan` *x y* [Function]

`greaterThan` *x y* [Function]

`lessThanOrEqual` *x y* [Function]

`greaterThanOrEqual` *x y* [Function]

These functions take Lpp Number objects for arguments *x* and *y* and return non-`nil` if the comparison is true and `nil` otherwise. For each of these functions respectively non-`nil` is returned if *x* is less than *y*, *x* is greater than *y*, *x* is less than or equal to *y*, *x* is greater than or equal to *y*.

Note that the equality of Numbers can be checked using `equal` and more efficiently using `eql` See Section 6.5 [Equality Predicates], page 29.

8.4 Arithmetic Operations

Basic arithmetic can be done on Lpp Numbers with the following.

`plus` *x y* [Function]

`minus` *x y* [Function]

`times` *x y* [Function]

`divide` *x y* [Function]

In these functions *x* and *y* must each be Lpp Numbers or `int` types and the returned result is a new Lpp Number. Their names describe what they do. It it worth mentioning however that `divide` does not truncate its value in any way, use the `truncate` function for that See Section 8.5 [Component Extractions on Numbers], page 36. If divide is given two number that do not divide evenly it will return an Lpp Ratio of the form `x/y` after it has been rationally canonicalized See Section 8.1 [Number Types], page 33. For example

```
divide(L(12), L(4)) => 3          // evenly divides
divide(L(13), L(4)) => 13/4       // doesn't so returns ratio
divide(4, L(12)) => 1/3           // returns canonicalized ratio
truncate(L(13), L(4)) => 3        // Use truncate function to truncate
```

`inc` *x* [Function]

`inc` *x delta* [Function]

`dec` *x* [Function]

`dec` *x delta* [Function]

These functions take an Lpp Number *x* and increments it in the case of `inc` and decrements it in the case of `dec`. The argument *x* is returned with the new value. Normally the increment or decrement is by 1, however if an optional second argument *delta* is given it must be an Lpp Number or `int` and the increment or decrement is by *delta* instead of 1.

Both `inc` and `dec` modify the object `x` so that most of the time a new object is not created. However there are cases where the a new object is created. This is when there is an object transition, for example from a SmallInteger to a BigInteger See Section 8.1 [Number Types], page 33. So in the following

```
inc(x, 1024);              // OK but ...
x = inc(x, 1024);          // this hadles object transition cases
```

if there is no transition expected then the first is fine. If a transition is possible then the second must be used and is the recommended form most of the time.

When there is an object transition Lpp automatically garbage collects the defunct object and returns the new incremented or decremented object. So in this case if the second code line above were not used then the new object would be lost and becomes a potential memory leak and `x` would point to deallocated memory.

`negate` *x* [Function]

The `negate` function simply negates the Lpp number object that `x` points to. While this is similar to `inc` and `dec` in that it modified the object `x`, the object `x` is never transitioned to another type of object and hence does not have the transitioning concern. In the following

```
negate(num);                    // Always OK
num = negate(num);              // Does the same thing, but unnecessary
other = negate(num);
```

the first is always safe to do and recommended, however the second works also but in unnecessary. The third would both negate `num` and set `other` to the negated `num`.

`gcd` *x y* [Function]

The `gcd` function returns the greatest common divisor of `x` and `y` each of which must be an Lpp Integer or `int` type. The result is always a positive Lpp Integer. Here are some examples

```
gcd(108, L(117)) => 9
gcd(L(123), -48) => 3
gcd(gcd(a, b), c);       // Returns the gcd of a b and c
```

8.5 Component Extractions on Numbers

`numerator` *x* [Function]

`denominator` *x* [Function]

Given *x*, an Lpp Rational number or `int`, the `numerator` function returns the numerator of *x* and `denominator` returns the denominator. Each function returns an Lpp Integer. If an integer n is given as the argument then it is treated as if it was n/1 so that `numerator` returns n and `denominator` returns 1. For example

```
let ratio = divide(23, -198);
numerator(ratio) => -23
denominator(ratio) => 198
numerator(numerator(ratio)) => -23
denominator(numerator(ratio)) => 1
```

`numeratorOf` *x* [Function]

`denominatorOf` *x* [Function]

Since Lpp maintains Ratio objects as pairs of Lpp Integer objects it allows the access of the actual numerator and denominator objects. The functions `numeratorOf` and `denominatorOf` are exactly the same as `numerator` and `denominator` respectively except for the fact that they do not make a copy but instead return the actual Integer object. This is not necessary in Lisp but in C++ this allows operations on the parts of a Ratio without having to worry about deallocating the copies. This is a useful efficiency; however the programmer has the responsibility to not inadvertently change parts of a Ratio object which could break the rational canonicalization rule and even break the code. For example

```
let ratio = readFromString("23/198");
times(numerator(ratio), denominator(ratio));      // Oops, memory leak
times(numeratorOf(ratio), denominatorOf(ratio));  // No leak
dec(numerator(ratio));          // OK, doesn't change ratio
dec(numeratorOf(ratio));        // Oops, does change ratio ...
                                // and breaks canonicalization rule,
                                // should now be 1/9 instead of 22/198
```

`truncate` *x* [Function]

`truncate` *x y* [Function]

`truncateCons` *x* [Function]

`truncateCons` *x y* [Function]

The `truncate` function with a single argument *x* converts it to an integer by truncating it toward zero. So if an integer n is given as the argument then it just returns a copy of n. If given a ratio as the argument then it is truncated as described here. If given two arguments x and y then `truncate` acts as if it is given a single ratio argument of x/y. The `truncateCons` function returns a `cons` whose `car` is what `truncate` would have returned and whose `cdr` is the remainder. The arguments to `truncate` and `truncateCons` can be `int` values or Lpp Rational numbers. Here are some examples

```
truncate(L(5)) => 5
truncate(divide(L(13), L(4))) => 3
truncate(readFromString("13/4")) => 3
truncate(13, L(4)) => 3
truncateCons(readFromString("13/4")) => (3 . 1)
let a = readFromString("5/3"); let b = readFromString("-2/3");
truncate(a, b) => -2
truncateCons(a, b) => (-2 . 1/3)
```

`floor` *x* [Function]

`floor` *x y* [Function]

`floorCons` *x* [Function]

`floorCons` *x y* [Function]

The `floor` family of functions above are the same as the `truncate` family above except that `floor` truncates toward negative infinity. For example

```
floor(L(5)) => 5
floor(divide(L(13), L(4))) => 3
floor(readFromString("13/4")) => 3
floor(13, L(4)) => 3
floor(-13, 4) => -4
floorCons(readFromString("13/4")) => (3 . 1)
floorCons(readFromString("-13/4")) => (-4 . 3)
let a = readFromString("5/3"); let b = readFromString("-2/3");
floor(a, b) => -3
floorCons(a, b) => (-3 . -1/3)
```

`rem` *x y* [Function]

`mod` *x y* [Function]

The `rem` function returns the same result that `truncateCons` of the same two arguments returns in its `cdr`. The `mod` function returns the same result that `floorCons` of the same two arguments returns in its `cdr`. For example comparing with the above examples of `truncateCons` and `floorCons`

```
rem(13, 4) => 1
mod(13, 4) => 1
mod(-13, 4) => 3
```

8.6 Efficient Specific Number Functions

When performing operations on Lpp numbers there is a very small cost in efficiency to dispatch on the types of numbers that are passed as argments to a number function. For example, in the following

```
plus(x, y);
```

the `x` and `y` arguments could be two SmallInteger objects, a SmallInteger and a BigInteger object, a BigInteger and a SmallInteger object, a SmallInteger and a Ratio object, a BigInteger and a Ratio object ... etc See Section 8.1 [Number Types], page 33.

This cost, albeit small, can be completely eliminated by casting specific types on the number arguments[2]. This can only work when the program knows absolutely what the specific types of numbers are occurring. For example if in a program segment it is absolutely known that only BigInteger objects are occurring then the following

```
plus(asThe(BigInteger, x), asThe(BigInteger, y));
```

would eliminate any dispatching cost.

This situation is rare and it is recommended that the cost is so small in proportion to the algorithms involved that this casting should routinely be avoided; it is simply not worth it.

[2] As of this writing, number functions only dispatch on the specific number types: SmallInteger, BigInteger and Ratio.

9 Characters

There is only one set of Lpp Character objects where each corresponds to one C++ character constant. So there is no overhead in generating Character objects and since there is only one object per character constant they can reliably be compared with `eq`. The initial Character objects are lazy generated, ie. only generated on demand since most characters are not even used in most program applications.

The character functions that deal with character attributes such as `alphaCharP`, `charUpcase`, etc. are only effective on the standard character set as defined by the Common Lisp specification.

9.1 Predicates on Characters

The function `characterp` can be used to determine if an object is a Character object, See Section 6.4 [Specific Data Type Predicates], page 28. In the rest of these predicates the given *character* argument must be a Character object; if it is not, an error is signaled.

`standardCharP` *character* [Function]

`standardCharP` returns `t` if *character* is a standard character and `nil` otherwise. See the Common Lisp specification for the definition of *standard characters*. For example

```
standardCharP(L('a')) => t
standardCharP(L(' ')) => t
standardCharP(L('\n')) => t
standardCharP(L('?')) => t
standardCharP(codeChar(2)) => nil
```

`alphaCharP` *character* [Function]

`digitCharP` *character* [Function]

`alphanumericp` *character* [Function]

`alphaCharP` returns `t` if *character* is an alphabet letter of either case and `nil` otherwise. `digitCharP` returns `t` if *character* is a 0-9 digit and `nil` otherwise. `alphanumericp` returns `t` if *character* is either a letter or digit and `nil` otherwise.

`upperCaseP` *character* [Function]

`lowerCaseP` *character* [Function]

`upperCaseP` returns `t` if *character* is an upper case letter and `nil` otherwise. `lowerCaseP` returns `t` if *character* is a lower case letter and `nil` otherwise.

9.2 Character Construction

`codeChar` *code* [Function]

`codeChar` given an integer `code` returns the Character object encoded as `code`. `code` can be an Lpp Integer, a C++ `int` or C++ character constant. For example in

```
let char1 = codeChar('a');
```

`char1` would be set to the Character object representing the C++ character constant 'a'. Note that the `L` operator can be used to do the same thing

```
let char1 = L('a');
```

but it is not symmetric in the sense that the `L` operator applied to an `int` will return an Lpp Integer instead of an Lpp Character as `codeChar` will.

`charCode` *character* [Function]

`charCodeL` *character* [Function]

`charCode` returns an `int` that encodes the given Lpp Character `character`. `charCodeL` does the same but returns an Lpp Integer object. For example using the variable `char1` above set to an Lpp Character object the following convert it to an Lpp Integer object, a C++ `int` and then to a C++ `char`

```
let char1Integer = charCodeL(char1);
int char1Int = charCode(char1);
int char1Int = cL(char1);
char char1Char = charCode(char1);
char char1Char = cL(char1);
```

Note that the results of the `cL` operator is exactly equivalent to the `charCode` function, See Section 3.2.2 [To C++ Primitive Types], page 15.

9.3 Character Conversions

`character` *object* [Function]

`character` returns the Character object that *object* can be coerced to or signals an error. Only Character, Integer, Symbol or String objects can be coerced to a Character object. For example

```
character(L('h')) => 'h'
character(L("h")) => 'h'
character(L(104)) => 'h'
character(S(h)) => 'h'
```

`charUpcase` *character* [Function]

`charDowncase` *characterz* [Function]

The given *character* must be a Character object. If *character* is a lower case letter `charUpcase` returns the corresponding upper case letter Character object. If *chacacter* is an upper case letter `charDowncase` returns the corresponding lower case letter Chacacter object. Otherwise both just return *character*.

10 Sequences

All Common Lisp functions that operate on sequences operate on either lists or vectors. Strings are currently the only vectors in Lpp.

Many sequence functions take an optional *test* argument that is a predicate function of two arguments that tests for the action of the function to take place, otherwise `eql` is used. For example in

```
remove(obj, sequence)
remove(obj, sequence, test)
```

The first `remove` will remove the elements of `sequence` that are `eql` to the object `obj`. But in the second `remove` where the function object `test` is given then that is used for the comparison instead of `eql`. The rule here is that where the test function object is of the form

```
test(obj, el)
```

The single object `obj` to compare is passed as the first argument and the second argument `el` being an element of the `sequence` to compare to is passed as the second argument. This ordering is important, for example if the test function was `lessThan` or a test function that compares two different object types by analyzing their components.

10.1 Simple Sequence Functions

`length` *sequence* [Function]

`lengthL` *sequence* [Function]

`length` returns the length of the sequence *sequence* as an `int`. `lengthL` is the same but returns an Lpp Integer object as the length. For example

```
length(0) => 0
let list1 = list(L("one"), L("two"), L("three"));
list(lengthL(list1)) => (3)
length(list1) + 1  => 4
length(nth(2, list1)) => 5
```

`elt` *sequence n* [Function]

`elt` returns element at index *n* of sequence *sequence*. It is an error if integer index *n* is outside the boundary of *sequence*. *n* can be either an `int` or Lpp Integer.

`setElt` *sequence n value* [Function]

`setElt` *sequence n value gc* [Function]

`setElt` sets the element at index *n* of sequence *sequence* to value *value* and *value* is returned. It is an error if integer index *n* is outside the boundary of *sequence*. *n* can be either an `int` or Lpp Integer. If an optional fourth argument *gc* is given and is a function of one argument it is applied to the element replaced. If argument *gc* is `t` then it defaults to the function `gc`. If *gc* is `nil` then nothing is applied. The optional fourth argument `gc` is ignored if *sequence* is a String.

`subseq` *sequence start* [Function]

`subseq` *sequence start end* [Function]

`subseq` returns a copy of the given sequence *sequence* from given *start* position index to the end of the sequence where 0 is the beginning of the sequence. If the optional argument *end* is given then that is used instead of the end of the sequence. The *end* position is not included in the returned sub-sequence. *start* and *end* can be either `int` or Lpp Integer objects. If *end* is an Lpp object and is received by `subseq` as `nil` then that is also interpreted as the end of the sequence. Some examples are

```
subseq(L("abcdefg"), 3) => "defg"
subseq(L("abcdefg"), 3, Nil) => "defg"
subseq(L("abcdefg"), 3, 5) => "de"
subseq(list(L(1), L(2), L(3)), 0, 1) => (1 2)
```

`copySeq` *sequence* [Function]

`copySeq` returns a copy of the sequence *sequence*. A fresh copy of the sequence is returned and it is guaranteed to be `equalp` to *sequence* but not `eq` to it. For example

```
let string1 = L("Hello");
let string2 = copySeq(string1);
equal(string1, string2) => t
eq(string1, string2) => nil
```

`reverse` *sequence* [Function]

`nreverse` *sequence* [Function]

`reverse` returns a copy of the sequence *sequence* with all of its elements reversed. `nreverse` does the same as `reverse` but a copy is not made and *sequence* is modified. For example

```
let list1 = list(L(1), L(2), L(3), L(4));
reverse(list1) => (4 3 2 1)
list1 => (1 2 3 4)
nreverse(list1) => (4 3 2 1)
list1 => (4 3 2 1)
let string1 = L("1234");
reverse(string1) => "4321"
string1 => "1234"
nreverse(string1) => "4321"
string1 => "4321"
```

10.2 Modifying Sequences

`remove` *x sequence* [Function]

`remove` *x sequence test* [Function]

`remove` returns a copy of sequence *sequence* with any element `eql` to *x* removed. If the optional third argument *test* is given it must be an Lpp Function of two arguments and it used in place of `eql` for the test. For example:

```
let list1 = list(L(1), L(2), L(3));
let list2 = remove(L(2), list1);
list1 => (1 2 3)
list2 => (1 3)
list1 = list(L("1"), L("2"), L("3"));
list2 = remove(L("2"), list1);
list1 => ("1" "2" "3")
list2 => ("1" "2" "3")
list2 = remove(L("2"), list1, L(stringEqual));
list1 => ("1" "2" "3")
list2 => ("1" "3")
```

`nremove` *x sequence* [Function]

`nremove` *x sequence test* [Function]

`nremove` *x sequence test gc* [Function]

`nremove` returns the sequence *sequence* with any element `eql` to *x* removed. Unlike `remove`, `nremove` is a destructive operation on *sequence*. If the optional third argument *test* is given it must be an Lpp Function of two arguments and it used in place of `eql` for the test. If test is given but `nil` then again `eql` is assumed.

The fourth argument *gc* tells `nremove` how the removed elements should be garbage collected. If sequence is a string then *gc* is ignored. If *gc* is nil then they are not collected.

If *gc* is `t` then they are fully collected. If *gc* is an Lpp Function of one argument then it is called on the removed element.

If sequence is a `list` then *gc* is applied to the removed cons and a `t` value is equivalent to passing in the `gcConsElement` function. If you wanted to collect just the cons element but not its contents you would pass in the `gc` function as the *gc* argument. Here are some examples for lists

```
let list1 = list(L(1), L(2), L(3));
let list2 = nremove(L(2), list1);
list1 => (1 3)
list2 => (1 3)
let listKeep = list(L(1), L(2), L(3));
let list1 = list(L(1), listKeep, L(3));
list1 => (1 (1 2 3) 3)
let list2 = nremove(listKeep, list1, 0);
list1 => (1 3)
list2 => (1 3)
listKeep => (1 2 3)
list1 = list(L("1"), L("2"), L("3"));
list2 = nremove(L("2"), list1, Nil, True);  // Strings are not eql!
list1 => ("1" "2" "3")
list2 => ("1" "2" "3")
list2 = nremove(L("2"), list1, L(stringEqual), True);
list1 => ("1" "3")
list2 => ("1" "3")
```

Here are some examples of `remove` and `nremove` for strings

```
let string1 = L("Hello World");
remove(codeChar('o'), string1) => "Hell Wrld"
string1 => "Hello World"
nremove(codeChar('o'), string1) = "Hell Wrld"
string1 => "Hell Wrld"
```

10.3 Searching Sequences

`position` *obj seq* [Function]

`position` *obj seq start* [Function]

`position` *obj seq start end* [Function]

`positionL` *obj seq* [Function]

`positionL` *obj seq start* [Function]

`positionL` *obj seq start end* [Function]

`position` returns the position of the first object found in the sequence *seq* that is `eql` to the given object *obj*. If the object is not found in the sequence then a `-1` is returned. The `positionL` functions are the same but an Lpp Integer object is returned for the position and `nil` is returned if the object is not found. The optional *start* argument is a positive integer as either an `int` or Lpp Integer object and indicates at what position to start the search. The optional *end* argument is also a similar positive integer and it indicates at what position to end the search. In this case the search is from *start* to one element before *end*. The *end* argument can also be `nil` defaulting *end* to be the length of the sequence, which would cause a search to the end of the sequence.

`positionTest` *obj seq test* [Function]

`positionTest` *obj seq test start* [Function]

`positionTest` *obj seq test start end* [Function]

`positionTestL` *obj seq test* [Function]

`positionTestL` *obj seq test start* [Function]

`positionTestL` *obj seq test start end* [Function]

The `positionTest` and `positionTestL` functions are exactly the same as the `position` and `positionL` functions except that instead of testing for the object *obj* in the sequence using `eql` the function *test* is used. *test* must be a function of two arguments the first being the object *obj* and the second an element of the sequence *seq* and must return non-`nil` for when an object satisfies the test.

Here are some examples

```
let string1 = L("abcabc");
position(L('a'), string1) => 0
position(L('c'), string1) => 2
position(L('a'), string1, 1) => 3
positionL(L('a'), string1, 1, 3) => nil

let list1 = list(L(1), L(2), L("three"), L(4));
position(L(4), list1) => 3
position(L("three"), list1) => -1
positionL(L("three"), list1) => nil
positionTest(L("three"), list1, L(equal)) => 2
```

10.4 Efficient Sequence Functions

All sequence functions must dispatch dynamically on the specific type of sequence passed into the sequence function call. This is a very small overhead especially in Lpp. However for absolute efficiency any sequence function can be typed for a specific sequence type and this dispatching overhead will not be incurred. For example in

```
length(sequence)
```

if we only expect `sequence` to be an Lpp String then

```
length(the(String, sequence))
```

would cause no sequence dispatching to take place. Or the user could define something like

```
inline let length_string(let sequence) {
  return length(the(String, sequence);}
inline let length_list(let sequence) {
  return length(theOrNil(Cons, sequence);}
```

and then use `length_string` when it is expected that `sequence` will be a string and `length_list` when expecting a list.

Since the whole idea of these specific type calls is absolute efficiency, when calling a sequence function using this method and the function overloads arguments as C++ types or Lpp types then only the C++ types are allowed. For example if the normal sequence function can take an `int` or an Lpp Integer then in the specific type function arguments only the `int` will work

```
subseq(asThe(String, sequence), 3)          // Works
subseq(asThe(String, sequence), iL(L(3)))   // Works
subseq(asThe(String, sequence), L(3))       // Will not even compile
```

notice that the last is not allowed by this rule and will not work or even compile.

Frankly, unless absolute efficiency is desired, the overhead on sequence dispatching is so small that this specific dispatching method is rarely worth doing.

11 Lists

In Lpp, as with Lisp, lists are built up using *cons* cells and a *list* is defined as either a cons or `nil`. In Lpp a *cons* cell is an Lpp object with just two members. The *car* is thought of as the left member and the *cdr* as the right member. Ordinary lists are built up with conses, where for each cons its car contains the contents of the list element and its cdr contains a pointer to another cons cell. Such a list is usually terminated with `nil` in the cdr of the last cons cell of the list. Lists are printed with the Lpp printing functions exactly as they are in Common Lisp. For example the list of the three symbols `a b c` is printed as

```
(a b c)
```

A list with a non-nil in the last cdr is called a *dotted list*. A one element list with a non-`nil` in the cdr is called a *dotted pair*. Note that a one element list is a single cons cell. Dotted pairs are useful in association lists or anywhere an efficient container for associated dynamic typed variables is needed. They are called dotted pairs because they are printed in Lisp and by the Lpp printing functions as

```
(x . y)
```

where `x` is the car and `y` is the cdr. A dotted list would be printed like

```
(a b c . d)
```

A list can also be circular. For example in the dotted list above if instead of `d` in the cdr of the last cons it contained the list itself then it would be circular. A list that is not circular or dotted is called a *proper* list.

As can be seen from the flexibility of cons cells as dotted pairs and lists, cons cells are also useful for building trees. For the purpose of this manual a *tree* is any Lpp object that may be an atom (a non-cons object) or a cons whose car and/or cdr may contain another cons nested to any level. Lists can also be used as *sets* with various set operations.

11.1 Conses

As mentioned in the preceding section conses can serve as fundamental building blocks of lists, association lists, dotted lists, trees and sets. A single cons is created with the `cons` function.

`cons` *x y* [Function]

The `cons` function returns a new cons whose car is *x* and cdr is *y*. For example:

```
cons(S(a), S(b)) => (a . b)
cons(S(a), list(S(b), S(c), S(d))) => (a b c d)
```

The accessors of a cons are the `car` and `cdr` functions. The car and cdr of a cons can be set with the `rplaca` (replace car) and `rplacd` (replace cdr) functions.

`car` *list* [Function]

This returns the car of *list* which must be either a cons or `nil`. The `car` of `nil` returns `nil`. For example:

```
car(cons(S(a), S(b))) => a
car(list(L(1), L(2), L(3))) => 1
```

`cdr` *list* [Function]

This returns the cdr of *list* which must be either a cons or `nil`. The `cdr` of `nil` returns `nil`. For example:

```
cdr(cons(S(a), S(b))) => b
cdr(list(L(1), L(2), L(3))) => (2 3)
```

`caar` *cons* [Function]

`cadr` *cons* [Function]

`cdar` *cons* [Function]

`cddr` *cons* [Function]

These functions are a convenience but also are important as commonly passed function objects. They perform successive car and or cdr operations of the cons argument *cons* and are defined by the following equivalents

```
caar(x) == car(car(x))
cadr(x) == car(cdr(x))
cdar(x) == cdr(car(x))
cddr(x) == cdr(cdr(x))
```

`rplaca` *x y* [Function]

The argument *x* must be a cons and *y* can be any Lpp object. This changes the car of *x* to *y* and returns the cons *x* after it has been modified. For example:

```
let c = cons(L(1), L(2));
rplaca(c, S(a)) => (a . 2)
c = list(S(a), S(b), S(c));
rplaca(c, L(1)) => (1 b c)
```

`rplacd` *x y* [Function]

The argument *x* must be a cons and *y* can be any Lpp object. This changes the cdr of *x* to *y* and returns the cons *x* after it has been modified. For example:

```
let c = cons(L(1), L(2));
rplacd(c, S(a)) => (1 . a)
rplacd(c, list(L(2), L(3))) => (1 2 3)
```

11.2 List operations

The following are operations on lists.

`nth` *n list* [Function]

This returns the nth element indicated by *n* of the list *list* where the car of *list* is the 0th element. *n* may be an Lpp Integer or an `int` and must be positive. If the length of the list is not greater than *n* then `nil` is returned. For example:

```
let list1 = list(S(zero), S(one), S(two));
nth(0, list1) => zero
nth(L(2), list1) => two
nth(10, list1) => nil
```

`nthcdr` *n list* [Function]

This returns the result of applying the `cdr` function *n* times on *list*. *n* may be an Lpp Integer or an `int` and must be positive. If the length of the list is not greater than *n* then `nil` is returned. For example:

```
let list1 = list(S(zero), S(one), S(two));
nthcdr(1, list1) => (one two)
nthcdr(10, list1) => nil
```

`first` *list* [Function]

`second` *list* [Function]

`third` *list* [Function]

`fourth` *list* [Function]

`fifth` *list* [Function]

`sixth` *list* [Function]

`seventh` *list* [Function]

`eighth` *list* [Function]

`ninth` *list* [Function]

`tenth` *list* [Function]

These functions are provided for convenient accessing of predetermined list element positions instead of `nth` and are more efficient. `first` returns the same as `nth` with a first argument of 0, `second` with a first argument of 1, etc.

`list` *args* [Function]

`list` constructs and returns a list of *args* which can be 1 or more arguments of type `let`. Up to 10 arguments may be given. For example:

```
list(L("zero"), S(one), L(2)) => ("zero" one 2)
list(S(zero), S(one), list(L(2), L(3)))
        => (zero one (2 3))
```

If more than 10 is needed `listEM` should be used.

`EM` [Constant]

`listEM` *args* [Function]

`listEM` is like `list` except that *args* can be any number of type `let` arguments. A global constant `EM`, abbreviation for "End Marker", terminates the arguments. For example:

```
listEM(L("zero"), S(one), L(2), EM) => ("zero" one 2)
let n = L(1);
listEM(n, n, n, n, n, n, n, n, n, n, n, EM)
        => (1 1 1 1 1 1 1 1 1 1 1)
```

`listSEM` *args* [Function]

`listSEM` is like `listEM` except that *args* can be any number of type `char*` arguments. For example:

```
listSEM("zero", "one", "two", EM) => ("zero" "one" "two")
```

`last` *list* [Function]

`last` *list n* [Function]

`last` returns the last *cons* of the list *list*. With the optional integer argument *n* it returns the tail of the list consisting of the last *n* conses of *list*. The *n* argument may be either an Lpp Integer or an `int`.

`endp` *object* [Function]

`endp` is false if *object* is a cons, true if `nil`, and signals an error for all other values. It is useful for checking for the end of a proper list.

`listLength` *list* [Function]

`listLengthL` *list* [Function]

`listLength` is the same as `length` applied to a list but is guaranteed to return `-1` if the list *list* is a circular list. `listLengthL` is the same as `lengthL` but returns `nil` if *list* is a circular list. `listLengthL` can also be used as a function object.

`copyList` *list* [Function]

`copyList` returns a list that is `equal` to *list* but not `eq`. Only the top level of the list is copied, that is it copies in the cdr direction and not the car direction.

`copyAlist` *list* [Function]

`copyAlist` is meant for copying association lists, See Section 11.4 [Association Lists], page 53. It is just like `copyList` except that for each element a cons is replaced in the copy by a new cons with the same car and cdr.

`copyTree` *object* [Function]

`copyTree` is for copying trees of conses. If *object* is not a cons it is just returned. If *object* is a cons the whole tree of conses in its car and cdr is copied recursively and returned. In this process conses are copied but non-conses are placed in the new cons as is.

`nconc` *list1 list2* [Function]

`nconc` concatenates *list1* to *list2* and return the concatenated list. Its arguments must be lists or `nil`. The arguments are changed rather than copied.

`listConcat` *list1 list2* [Function]

`listConcat` is just like `nconc` except that its arguments `list1` and `list2` are copied and the copies are concatenated and returned.

For the next two macros `push` and `pop` the argument *place* is any variable that contains a list. Note that by the definition of a list `nil` is also considered a list.

`push` *item place* [Macro]

item is any object of type `let`. After the `push` macro executes *place* will be a list with *item* as the car of the list and the list that was in *place* as the cdr.

`pop` *var place* [Macro]

`var` is any variable of type `let`. After the `pop` macro executes *var* will contain the car of *place* and *place* will contain the cdr.

11.3 Using Lists as Sets

`member` *item list* [Function]

`member` *item list test* [Function]

`member` searches the list *list* for an object `eql` to the Lpp object *item*. If none is found `nil` is returned otherwise the tail of *list* beginning with the element `eql` to *item* is returned. *list* is searched at the top level only. Note that `member` can also be used as a predicate.

If the optional argument `test` is given it must be an Lpp Function object of two arguments that performs the test in place of `eql`.

For example:

```
member(L(2), list(L(1), L(2), L(3))) => (2 3)
member(L(2), list(S(one), S(two), L(three))) => nil
let snums = listSEM("1", "2", "3", EM);
member(L("2"), snums) => nil
member(L("2"), snums, L(stringEqual)) => ("2", "3")
```

11.4 Association Lists

An *association list* is a list of pairs where each pair is an association of a *key* to a *datum*. The pair is represented as a cons where the car is the *key* of the pair and the cdr is the *datum*.

`assoc` *key list* [Function]

`assoc` *key list test* [Function]

`assoc` searches for a cons in *list* whose car is `eql` to *key*. If found, the cons is returned otherwise `nil` is returned. If the optional third argument *test* is given it must be an Lpp Function of two arguments and `assoc` will use it to do the test in place of `eql`. For example:

```
let list1 = list(cons(L(1), L(2)),
                 cons(L(2), L(4)), cons(L(3), L(6)));
assoc(L(2), list1) => (2 . 4)
assoc(L(4), list1) => nil
let list1 = list(list(L("1"), L(2)),
                 list(L("2"), L(4)), list(L("3"), L(6)));
assoc(L("2"), list1) => nil
assoc(L("2"), list1, L(stringEqual)) => ("2" 4)
```

12 Hash Tables

A hash table is an Lpp object that can efficiently map any Lpp object to any other Lpp object. Each hash table has a set of entries each which associates a particular *key* with a *value*. Entries can be created, removed, modified and found given the *key*. Since an optimal hash function and algorithm is used, finding the *value* is very fast even if there are many entries.

12.1 Hash Table Functions

`makeHashTable` [Function]

`makeHashTable` *test* [Function]

`makeHashTable` *test size* [Function]

`makeHashTable` returns a new Lpp hash table object. If the optional argument *test* is given it should be a predicate function object of two arguments that returns `nil` unless the two arguments representing hash table keys are considered equal by the definition of the function. Or *test* may be one of the symbols: `eq`, `eql` or `equal` which would correspond to the function of that symbol name. If *test* is not given or `nil` it is assumed to be the `eql` function.

The optional argument *size* if given should be an positive integer argument specifying the size of the hash table. *size* may be of type `int` or Lpp Integer. If *size* is not given or `nil` then a default size of approximately 500 is used. The user is encouraged here to either use the default size or study how to compute the optimal size of hash tables (See Knuth "Art of Computer Programming"). Although all sizes will work under Lpp some sizes will result in better distributions.

`puthash` *key table value* [Function]

`puthash` creates an entry for *key* with value `value` in the hash table *table*. If the *key* entry already exists in *table* then its value is replaced with *value*. *value* is returned.

`gethash` *key table* [Function]

`gethash` *key table default* [Function]

`gethash` finds the entry for *key* in the hash table *table* and returns the entry value. If not found it returns `nil`. If the optional third argument `default` is specified and the `key` is not found *default* is returned instead of `nil`.

Here is an example:

```
let table = makeHashTable();
let max = L(100); let min = L(0);
puthash(S(red), table, list(max, min, min));
puthash(S(green), table, list(min, max, min));
gethash(S(blue), table) => nil
gethash(S(red), table) => (100 0 0)
```

`remhash` *key table* [Function]

`remhash` removes the entry for *key* in hash table *table*. This is also a predicate that returns `t` if there was an entry and `nil` if not.

`clrhash` *table* [Function]

`clrhash` removes all entry from the hash table *table* and returns *table*.

`hashTableCount` *table* [Function]

`hashTableCountL` *table* [Function]

`hashTableCount` returns the number of entries in hash table *table* as an `int`. `hashTableCountL` does the same but returns an Lpp Integer. A hash table starts out with zero entries.

`maphash` *function table* [Function]

For each entry in hash table *table* `maphash` calls function *function* on two arguments: the key of the entry and the value of the entry. For example the following `maphash` would display the key and value for all entries in hash table `ht`

```
let printKeyValue(let key, let value) {
  princ("  key = "); prin1(key);
  princ("    value = "); prin1(value); terpri();
  return Nil;}

maphash(L(printKeyValue), ht);
```

12.2 Primitive Hash Function

The primitive hash function `sxhash` is used internally by the Lpp hash functions but is made public so that the user can create other hashing structures.

`sxhash` *object* [Function]

`sxhashL` *object* [Function]

`sxhash` returns a hash code positive integer for the Lpp object *object* as an `int`. The returned integer is such that for given objects `obj1` and `obj2`

```
equal(obj1, obj2) => sxhash(obj1) == sxhash(obj2)
```

So for example two different strings "abc" will return the same `sxhash` integer. The function `sxhashL` is exactly the same except that it returns an Lpp Integer object and it is guaranteed that for a given object `obj`

```
sxhash(obj) == iL(sxhashL(obj))
```

13 Strings

Lpp Strings are considerably more efficient than ordinary C++ strings in algorithms that require the length of the string since the length is held as a slot in the String object. Ordinary C++ strings must be searched for the 0 end character. Also, null characters (0 character code) are allowed inside Lpp Strings and not inside ordinary C++ strings. When converting from Lpp Strings to C++ strings the programmer must be aware of this. If there is a 0 in an Lpp String then the converted ordinary C++ string will look like it's been truncated. Rarely is this a problem since either the programmer knows that in an application 0's may be stored in Strings in which case he only uses Lpp Strings (in such a case something like Lpp Strings must be used anyway) or the programmer's application only needs to convert ordinary C++ strings to Lpp Strings but not vice verse.

13.1 String Comparison

`stringEqual` *s1 s2* [Function]

`stringEqual` compares Lpp String *s1* to Lpp String *s2* and returns `t` if they are the same length and all characters of *s1* are the same characters of *s2*. Otherwise `nil` is returned. For example:

```
stringEqual(L("foo"), L("Foo")) => nil
stringEqual(L("foo"), L("foo")) => t
```

`stringEQUAL` *s1 s2* [Function]

`stringEQUAL` is the same as `stringEqual` except that character case is ignored. For example

```
stringEQUAL(L("foo"), L("Foo")) => t
```

13.2 String Construction and Manipulation

`makeString` *size* [Function]

`makeString` *size char* [Function]

`makeString` returns a new string of given *size* which can be an `int` or Lpp Integer. The string characters are not initialized unless the optional argument *char* is given in which case the whole string is initialized to characters of *char* which must be an Lpp Character object.

`stringConcat` *s1 s2* [Function]

`stringConcat` returns a new Lpp String which is a concatenation of Lpp Strings *s1* and *s2*. For example:

```
stringConcat(L("one "), L("two")) => "one two"
```

14 Input Output

Lpp provides typical Common Lisp functions for doing input/output of Lpp objects such as `print`, `read` etc. In addition to these Lpp objects can also be output to C++ streams automatically with no extra work.

14.1 Printed Representation of Lpp Objects

All predefined Lpp objects have `prin1` and `princ` methods that print the object as you would expect in Common Lisp. For example the following code for printing some Lpp objects

```
prin1(L("Hello world"); terpri();
princ(L("Hello world");
print(cons(S(a), S(b)));
```

would produce on cout

```
"Hello world"
Hello world
(a . b)
```

As described in this manual, See Section 3.4 [Accessing Type Meta-Objects], page 17, the programmer can dynamically set the `prin1` and `princ` methods of his objects. All Lpp objects whose `prin1` and `princ` methods have not been set have default print methods which prints the object as

```
<description  address>
```

where *description* describes the type of the object and *address* is the address in memory where the object is allocated.

`defaultPrin1Address` [Variable]

The `defaultPrin1Address` global variable when non-`nil` will cause the default printing of objects to print as

```
<description>
```

that is leaving out the *address* part. This is useful for things like regression tests on program output where the same output is expected from one run to another.

The value of `defaultPrin1Address` defaults to `t`.

14.2 Reading

Lpp objects can be read and constructed from C++ input streams.

`read` [Function]

`read` *stream* [Function]

`read` reads one Lpp object from the C++ input stream *stream* and returns that object. The syntax of such objects on *stream* is the same as Common Lisp except with no special dispatching characters such as "#" or quote. Lpp `read` does ignore semicolon comments to the end of line however. If the *stream* argument is not provided `cin` is assumed. For example suppose that we type (to `cin`) the following:

```
red
"Hello world"
(1 two 3)
```

Then calling `read` three times will produce:

```
let symbol1 = read();
let string1 = read();
let list1 = read();
symbolp(symbol1) => t
stringp(string1) => t
listp(list1) => t
numberp(first(list1)) => t
list(symbol1, string1, list1)
        => (red "Hello world" (1 two 3))
```

`readFromString` *string* [Function]

The argument *string* is either a `char*` string or an Lpp String object. `readFromString` is the same as read but uses the string *string* instead of a C++ stream. For example:

```
let symbol1 = readFromString("red");
let string1 = readFromString("\"Hello world\"");
let list1 = readFromString("(1 two 3")
symbolp(symbol1) => t
stringp(string1) => t
listp(list1) => t
numberp(first(list1)) => t
list(symbol1, string1, list1)
        => (red "Hello world" (1 two 3))
```

14.3 Printing

The Lpp printing functions emulate Common Lisp printing functions in that there are two ways to print Lpp objects to C++ streams, with or without escape characters. All Lpp objects automatically get two type dispatching printing functions for doing this. The user can redefine these dispatching functions if need be, See Section 3.4 [Accessing Type Meta-Objects], page 17.

`prin1` *object* [Function]

`prin1` *object stream* [Function]

`princ` *object* [Function]

`princ` *object stream* [Function]

`print` *object* [Function]

`print` *object stream* [Function]

All of these functions print the Lpp object *object* to the C++ output stream *stream*. If *stream* is omitted `cout` is assumed. All of these functions return *object* as its value.

With `prin1` escape characters are used as appropriate while `princ` prints *object* with no escape characters. Roughly speaking the output of `prin1` is suitable for `read` and the

output of `princ` is intended to look good to people. `print` is just like `prin1` except that the printed representation of *object* is preceded with a newline and followed by a space. Some examples will make this clear, the following code

```
let obj = L("one two three");
prin1(obj);
print(object);
princ(obj);
```

would produce on `cout`

```
"one two three"
"one two three" one two three
```

Note that the last output produced by `princ` when read back using `read` would produce the Symbol `one` instead of the String "one two three".

C++ output stream operators for Lpp objects default to using `princ` so that the following code

```
let obj = L("one two three");
cout << "Object = " << obj << endl;
```

would produce on `cout`

```
Object = one two three
```

`O` *object* [Macro]

The `O` macro allows the Lpp object argument *object* when using C++ output stream operators to print using `prin1`. Contrast the following code and results using the `O` macro with the previous example

```
let obj = L("one two three");
cout << "Object = " << O(obj) << endl;
```

would produce on `cout`

```
Object = "one two three"
```

Note that it only makes sense to use the `O` macro with C++ stream operators. The results of using the `O` macro elsewhere is undefined.

`terpri` [Function]

`terpri` *stream* [Function]

`terpri` outputs a newline to the C++ output stream *stream*. If the *stream* argument is omitted `cout` is assumed. `terpri` returns `nil`.

`finishOutput` [Function]

`finishOutput` *stream* [Function]

`finishOutput` attempts to ensure that all output sent to the C++ output stream *stream* has reached its destination and only then finally returns `nil`. If the *stream* argument is omitted `cout` is assumed.

`pprint` *object* [Function]

`pprint` *object stream* [Function]

`pprint` *object stream start end* [Function]

`pprint` is just like `print` except that the space is omitted after *object* is printed to the C++ output stream *stream* and the output is pretty. If the *stream* argument is omitted `cout`

is assumed. If the *start* and *end* arguments are included the output will be pretty printed between columns *start* and *end*. *start* and *end* must be `int`s. All of these functions return *object* as its value.

Pretty means that the object is printed using extra white space to make it easily human readable. For example nested lists are indented to make the tree structure apparent.

`prin1ToString` *object* [Function]

`princToString` *object* [Function]

Both of these functions return an Lpp String with the printed representation of *object*. `prin1ToString` returns the `prin1` printed representation. `princToString` returns the `princ` printed representation. For example:

```
let obj = prin1ToString(list(L(1), L(2), L(3)));
prin1(obj) => "(1 2 3)"
```

Typically `prin1ToString` and `princToString` are used to print relatively short expressions or values to strings. The maximum length of such a string allowed by default is 4096 characters which is determined by the define variable LPP_PRINTOSTRING_MAX. If more characters are desired for some reason redefine LPP_PRINTOSTRING_MAX to a value bigger than 4096 before the include file `Lpp.hh` in the Lpp library compile stream.

`prin1Length` *object* [Function]

`prin1LengthL` *object* [Function]

`princLength` *object* [Function]

`princLengthL` *object* [Function]

`prin1Length` returns as an `int` the `prin1` printed representation length of the Lpp object *object*. `prin1LengthL` does the same but returns the length as an Lpp Integer. `princLength` and `princLengthL` are the counterparts for `princ`. Here are some examples:

```
prin1Length(S(hello)) => 5
prin1Length(L("hello")) => 7
princLength(L("hello")) => 5
prin1Length(L(-1234)) => 5
prin1Length(list(L(1), L(2))) => 5
```

15 File System Interface

Lpp provides a small collection of file system interface functions that are somewhat similar to corresponding Common Lisp functions. Since C++ provides a library of file stream functions these Lpp file system interface functions are not absolutely necessary. They do however make it easy to bring the file system into Lisp semantics. For example the `fileDirectory` function returns a Lisp list of file names.

`currentDirectory` [Function]

`currentDirectory` returns an Lpp String that represents the environment's notion of the *current directory* that the Lpp program is executing in. For example in Unix this is the current directory when `main` was executed. The returned String has a terminating directory identifier (if such exists) such that if the String were concatenated with a plain file name the result would be a complete file specification. For example suppose on a Unix system the current directory is `/home/me/lpp/lib` then

```
stringConcat(currentDirectory(), L("filex"))
                        => "/home/me/lpp/lib/filex"
```

`fileDirectory` *file* [Function]

Given a file specification *file* `fileDirectory` will return a list of names of the files in the directory if the given specification names a directory in the file system or a file specification relative to the current directory. An error is signaled if the the given `file` can not be opened as a directory. The *file* argument can be an Lpp String or a `char*` string. The returned list of names are Lpp Strings. Here are some examples

```
let dirList = fileDirectory(currentDirectory());
int dirLength = length(fileDirectory("/home/me/lpp/lib/test"));
let filtered = mapcan(L(filterFunction), fileDirectory("test"));
```

Note that `fileDirectory` is similar to the `directory` function in Common Lisp. But while the argument to `directory` is a file pattern string in Common Lisp the argument to `fileDirectory` is an exact directory name.

`probeFile` *file pred* [Function]

Given a file specification *file* and a predicate symbol *pred* returns `t` if what *pred* indicates is true of the given *file* and `nil` otherwise. The argument *file* is a file specification in the file system or a specification relative to the current directory. It can be a `char*` string or an Lpp String. The *pred* argument can be one of the symbols: `exist`, `readable`, `writable`, `executable`, `directory`, `regular`, `symbolicLink`.

If given the symbol `exist` *probeFile* returns `t` if the given file specification exists. If give `readable`, `writable` or `executable` it will return `t` if the file is readable, writable or executable respectively. If given `directory` it returns true if the file specification is a directory. If given `regular` it will return `t` if the file is a regular file. Note that a regular file can be other than a non-directory in some file systems, such as a Unix file system. For file systems that have symbolic links, if given the symbol `symbolicLink` it returns `t` if the file is a symbolic link.

For example the `directory` symbol can be used to recursively descend directories

```
void recursiveDescend(let directory) {
  dolist(file, fileDirectory(directory)) {
    // Do something with this file.
    if (probeFile(file, S(directory))) recursiveDescend(file);}}
```

Note that calling `probeFile` with the symbol `exist` is similar to the `probe-file` of Common Lisp, that is, it is a predicate for the existence of the file. But where `probe-file` in Common Lisp returns the true name of the file `probeFile` just returns `t` if the file exists. This was done on purpose in Lpp to avoid generating a String object that would just need to be garbage collected if all that we wanted to use `probeFile` for was as a predicate.

16 Errors

`error` *string args* [Function]

`error` takes a string argument *string* and 0 or more optional arguments *args* and reports an Lpp error message using *string* and enters an error handler. The *string* argument can be either an Lpp String or a `char*` string and it may contain `printf` style substitution specifiers that correspond to the optional *args* just as in `printf`. For example:

```
if (typeOf(object) != type(MyType)) {
  char* objectID = sL(prin1ToString(object));
  char* expectType = typeName(type(MyTupe));
  error("Object %s is not of type: %s", objectID, expectType);}
```

17 Miscellaneous Features

17.1 Identity Function

`identity` *object* [Function]

This function simply returns the object `object`. This function is useful for debugging or when a Function object is needed by some other function and no operation more than identity is needed. See the `maplist` example using `identity`, See Section 5.2.2 [Mapping], page 24.

17.2 Debugging Tools

Lpp provides debugging tools, primarily to make it easier to debug programs using Lpp in run time debuggers like `gdb` (*Gnu Debugger*).

`pdb` *object* [Function]

`pdc` *object* [Function]

`ppdb` *object* [Function]

These functions can be used in a debugger to print and inspect Lpp objects. Each takes one argument *object* which is any Lpp object or 0 (note that 0 prints as `nil`). The names of these functions have been kept short for easy typing in a debugger. In some debuggers such as `gdb` they can be evoked from a user defined macro which gives even more brevity. `pdb` uses `prin1`, `pdc` uses `princ` and `ppdb` uses `pprint`. For example assume we are debugging the following code in `gdb`

```
let list1 = list(L("One"), L(2), S(Three));
findJunk(first(list1));         // <--- Assume breakpoint here
```

and assume that we had placed a breakpoint at the `findJunk` call. Then after running and catching the breakpoint in `gdb`

```
(gdb) p pdb(list1)
("One" 2 Three)

(gdb) p pdb(car(list1))
"One"

(gdb) p pdc(car(list1))
One
```

Note the difference between the `pdb` and `pdc` printing of the String `"One"`, the `pdb` version is quoted and the `pdc` version is not. This reflects the standard Common Lisp difference between the way that Lisp objects such as strings print using `princ` versus `print1`. However the Lpp user is free to capitalize on this for his own objects for example by having one print method for standard C++ stream output (`princ`) and another for debugging inspection purposes. And since print methods can be set dynamically in the object's type meta-object the user can have any number of ways to print an object depending on the setting, See Section 3.4 [Accessing Type Meta-Objects], page 17.

17.3 Garbage Collection

Lisp implementations have the advantage that all objects are accessible through some path originating in the symbol tables and the whole accessible structure is an intrinsic part of Lisp. Thus, much of the extra data structure for doing garbage collection in Lisp exists for free and is more cogent than C++ as a starting point for garbage collection algorithms. Since Lpp does not have this structure available under C++ and since there are many reasonably good C++ garbage collection algorithms available, Lpp does not commit to any of these in principle nor provides any kind of automatic garbage collection.

Lpp does provide a facility for doing manual garbage collection much the same way `delete` works in standard C++. But it makes the garbage collection of complex Lpp data structures easier. All Lpp garbage collection functions only serve to deallocate the storage of some data and then just return `nil`.

`gc` *object* [Function]

`gc` garbage collects the Lpp object *object*.

Here is a trivial example:

```
let a = L(1);
let b = L(2);
let c = plus(a, b);  // assume c is needed from here on
gc(a); gc(b);        // and a and b are not
```

In general, for predefined Lpp object types the object and all of its components are collected except for some exceptions listed in the next paragraph. For user defined Lpp objects `gc` calls `delete` of the specific subtype class for *object*.

Since Lpp symbols have global import `gc` will do nothing if *object* is a Symbol. Usually you do not want to remove symbols. However in cases where you absolutely must the proper way to remove symbols and have them collected is to use `unintern` which removes the symbol from the system and garbage collects all of the associated parts, See Section 7.2 [Creating Symbols], page 31. If `gc` is called on a `cons` it collects the cons object only and not its `car` or `cdr`.

For garbage collecting cons structures `gcList` should be used for garbage collecting top level lists and `gcTree` for collecting tree structures. In these functions a *leaf* argument says what to do with the leaf nodes of a list or a tree. Leaf nodes are a car or cdr of a cons that is something other than a list (`nil` or another cons). However the main difference between `gcList` and `gcTree` is that with `gcList` the `car` of every cons in the top level list is also treated as a leaf node for the listed meanings below of the *leaf* argument. Note that because of this using `gcList` with a special user collection function as the leaf argument the user can effectively create his own special version of `gcTree`. This also implies that if the programmer wants to use `gcTree` on a top level list that has a cons in some of its cars he should use `gcTree` instead or else supply his own leaf function argument to collect such cars or otherwise take care of them.

The *leaf* argument can be one of three things with the following implied meanings:

```
nil => Do not garbage collect leaf nodes
t => Apply gc() to leaf nodes
Function f => Apply Function f to leaf nodes
```

`gcList` will handle all lists including circular and dotted lists. `gcTree` will handle all trees except trees that have circular or multiple references. However it will break with an explanation error message if it detects either circular or multiple references for most trees. The only cases where it doesn't detect is where there are circular or multiple reference paths where there are no leaves in the path. For example

```
let self = list(True);
rplaca(self, self);
```

Clearly this simple tree has a circularity with no leaf nodes in the circular path. This would fail with `gcTree`. But this is not usual and is probably not even useful. In such unusual trees the programmer must either create his own collection function or else store a non-cons in such a problematical cons slot and then call `gcTree`.

`gcList` *list leaf* [Function]

`gcList` will garbage collect the list *list* and leaf nodes depending on the *leaf* argument as described above. For this purpose the cars of each cons in the list will always be treated as leaf nodes even if they contain other cons objects.

Here are some examples.

```
let list1 = list(L("one"), L("two"), L("three"));
let list2 = copyList(list1);

// leaf argument is nil, so list1 is grabage collected
// but Strings "one", "two", "three" are left alone.
gcList(list1, Nil);

// leaf argument is t, so list2 is grabage collected
// and so are Strings "one", "two", "three"
gcList(list2, True);

list1 = list(myObject1, myObject2);

// leaf argument is the Function myCollector
// so list1 is garbage collected while the function myCollector
// is applied to the leaf node objects myObject1, myObject2
gcList(list1, L(myCollector));
```

`gcTree` *tree leaf* [Function]

`gcTree` will garbage collect the tree *tree* and leaf nodes depending on the *leaf* argument as described above. The *tree* argument can be a cons or a non-cons Lpp object. If it is a non-cons object then the *leaf* argument is applied to the object.

If there are circular or multiple references in the tree in paths that have one of more leaves then `gcTree` will break with an error message stating such. For any trees with circular or multiple references the programmer can either store a non-cons in the offending cons slots and then call `gcTree` or write a gc function specialized for such a data structure and call `gcList` passing the specialized function as the leaf argument.

Here are some examples using association lists which are in effect small trees:

```
let alist1 = list(cons(L("one"), L(1)),
                  cons(L("two"), L(2)),
                  cons(L("three"), L(3)));
let alist2 = copyAlist(list1);

// leaf argument is nil, so alist1 is grabage collected
// but Strings "one", "two", "three" are left alone
// and numbers 1, 2 and 3 are left alone.
gcTree(alist1, Nil);

// leaf argument is t, so alist2 is grabage collected
// and so are Strings "one", "two", "three"
// and numbers 1, 2 and 3.
gcTree(alist2, True);

alist1 = list(cons(L(1), myObject1),
              cons(L(2), myObject2));

// leaf argument is the Function myCollector
// so alist1 is garbage collected while the function myCollector
// is applied to the leaf node objects 1, 2, myObject1, myObject2
gcTree(alist1, L(myCollector));


It is easy to garbage collect raw cons objects and their parts.
For example assume the variable var contains a cons,
then

     gcTree(cdr(var));  // Collects just the whole cdr tree of var
     gc(var);  // Collects just the cons var and not the car or cdr

There are many combinations of ways to collect cons structures
and their parts.  But the following function fulfills a common case
```

gcConsElement *cons* [Function]

```
Given a cons in the cons argument gcConsElement will
garbage collect the whole car tree of cons using
gcTree and then garbage collect the cons itself.  It is
useful for completely collecting ordinary list elements that are
represented by cons structures and is used by some Lpp functions that
optionally perform a garbage collection on destructive list
operations, such as nremove, See Section 10.2 [Modifying Sequences], page 45.
```

18 Programming Cautions

This chapter is more for programmers that are used to Lisp implementations and need to be aware of some idiosyncrasies of C++ as it relates to Lpp.

18.1 Memory Leaks

Infamous to C and C++ are those annoying memory leaks. In Lisp a programmer does not have to worry about memory allocation but it is a different story in C++. When a program allocates memory and neglects to deallocate the memory when it is no longer needed then a potential memory leak exists. If a program has a memory leak in some code and then loops on that code then eventually the program will use up available memory and break with a memory allocation error.

In C++ one must either link with one of the many automatic garbage collector libraries or else each function must make sure which data it owns and manually deallocate the data's memory when it is no longer referenced. Lpp provides some manual garbage collection functions which help See Section 17.3 [Garbage Collection], page 65. But the programmer must still be aware of how potential memory leaks can occur.

All Lpp data is allocated from the heap and not the stack. So no Lpp data is automatically deallocated by virtue of exiting a function. Symbols and character objects though are not problematical. There is only one set of character objects that are shared by all programs. And symbols once interned are also shared by all programs. So for example in the following code

```
list1 = list(S(sym1), S(sym1), S(sym1))
```

the symbol `sym1` will only get allocated on its first internment. Or in other words, in the case of `list1` the first `S(sym1)` will intern the symbol `sym1` and allocate it (assuming it has not been interned previously) and then return the symbol. The second and third `S(sym1)` then will only return the already interned symbol `sym1`.

Usually the programmer will never have a need to deallocate symbols since the nature of their semantics is to be persistent within the full extent of the program. Symbols can be deallocated with `unintern` if it is absolutely necessary See Section 7.2 [Creating Symbols], page 31.

To be safe then, all other Lpp objects should be garbage collected when known to be no longer referenced or else simulate the Symbol semantics. For example if in a function, we reference a commonly used string like the following

```
let b = stringConcat(a, L("-something-common"));
```

and that function can be called repeated times, then the string should either be made static

```
static let common_string = L("-something-common");
let b = stringConcat(a, common_string);
```

or alternatively it can be manually garbage collected

```
let common_string = L("-something-common");
let b = stringConcat(a, common_string);
gc(common_string);
```

otherwise there will be no reference to such strings allocated and when the function exits the string objects would exist in memory until the program exits.

The manual garbage collection in this last case is a bad choice since we know that the same string will be used over and over again. The static variable containing the common string would be a better choice. But in a case where there is an intermediate value whose value we can not predict then a manual garbage collection would be more appropriate. The following for example would be a such a memory leak

```
let a = minus(d, times(b, c));
```

where b, c and d are input variables. The memory leak would be where the `times` returns a value. That value is not referenced anywhere after the minus is executed. So to prevent a memory leak here this should be coded instead as

```
let temp = times(b, c);
let a = minus(d, temp);
gc(temp);
```

The kinds of potential memory leaks mentioned above are relatively simple to find and fix. But in the real world where large complex data structures are passed around, memory leak bugs can be quite difficult to pin down. It is a classical problem of which function owns what data when. Lpp provides some assist to this by making it easier to sweep up large branches of data that are no longer referenced with functions like `gcList` and `gcTree` but the programmer must carefully design the growth and collapse of such complex structures so as to avoid memory leaks.

18.2 Order of Evaluation

In Lisp the order of evaluation of function arguments is strictly left to right. Sometimes this is important since successive argument evaluation may depend on earlier arguments. Unfortunately in C++ the order of evaluation is undefended. Therefore the programmer must not depend on this. For example in the following code

```
let a = L(5);
list(inc(a, a), dec(a));
```

the `list` function could return (10 9) or (8 4) depending on the C++ implementation. There is no standard guideline here other than to watch out for such order of evaluation problems and if necessary re-code in a different way.

Appendix A Appendix - read/print User Interface (rpUI)

Lpp comes bundled with rpUI, a read/print User Interface, that is implemented using the Lpp library. By providing a selection of simple menus rpUI makes it easy to create complete applications using Lpp. Read/print interfaces are also good for creating debugging interfaces to applications and regression testing procedures.

Two kinds of menus are available: First, *Static Numerical* menus present the same fix list of choices for the user with sequential integers prefixing each choice. The user types in an integer to make a selection. Second, *Dynamic Symbolic* menus just prompts for a symbol and a choice is made based on the symbol that the user enters.

`rpSnMenu` *spec* [Function]

This function returns an Lpp object representing a Static Numerical menu based on the specification *spec* provided. The menu object returned can be used in subsequent calls to `chooseRpMenu` where the user needs to make a menu selection. *spec* is a list of Lpp Strings. The first entry in the list will be the menu header. The rest of the list will specify sequential menu selections. When `chooseRpMenu` is called the user will be presented with the header followed by the choices prefixed with integers and then a prompt for an integer. If the user enters anything other than one of the integers corresponding to a choice the user is notified of a bad entry and the menu is presented again. If the user enters an integer corresponding to a choice that integer is returned from `chooseRpMenu` as in `int`.

`rpDsMenu` *spec* [Function]

This function returns an Lpp object representing a Dynamic Symbolic menu based on the specification *spec* provided. The menu object returned can be used in subsequent calls to `chooseRpMenu` where the user needs to make a menu selection. *spec* is a list of Lpp Strings. The first entry in the list will be the menu prompt. The rest of the list is alternating *symbol description* Strings where *symbol* will be the expected symbol that the user is expected to enter for that choice and *description* is a String describing the choice. When `chooseRpMenu` is called the user will be presented with the prompt. If the user enters anything other than one of the symbols corresponding to a choice the user is notified of a bad entry and then menu of choices with *symbol description* pairs is presented. If the user enters a symbol corresponding to a choice then an integer is returned from `chooseRpMenu` as an `int`. The integer corresponds to the position in the *spec* list of choices, where the first choice will return 0, the second 1 etc.

`chooseRpMenu` *menu* [Function]

This function is given the argument *menu* which is either an `rpSnMenu` or an `rpDsMenu`. The menu or prompt is presented as indicated above and when the user makes an entry an `int` of 0 to *n* is returned where there are *n - 1* choices in the menu.

The following simple program illustrates the use of both kinds of menus:

```
#include <rpMenu.hh>
#include <Lpp.hh>

main() {
  let menu1 = rpDsMenu
    (L("Enter command (or ? for help): "),
     listSEM("exit", "Exit program",
             "fix", "Fix me",
             "list", "List Sn menu", EM));
  let menu2 = rpSnMenu(L("Choose one of"),
                       listSEM("choice 0", "choice 1", EM));
  int choice; int doing = 1;
  while (doing) {
    switch (chooseRpMenu(menu1)) {
    case 0: doing = 0; cout << "Goodbye\n"; break;
    case 1: cout << "Thank you!\n"; break;
    case 2:
      choice = chooseRpMenu(menu2);
      cout << "\nChoice = " << choice << endl;}}}
```

When the above program is run it will produce the following dialog:

```
Enter command (or ? for help):  ?

Choose one of:
  exit = Exit program
  fix = Fix me
  list = List Sn menu
Enter command (or ? for help):  list

Choose one of
  0 = choice 0
  1 = choice 1
Enter 0-1: 1

Choice = 1

Enter command (or ? for help):  exit
Goodbye
```

Function Index

A

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

Z

Variable Index

Concept Index