The L Documentation

Matthieu Lemerre

This manual is for L, version 0.0.1

Published by nobody

Table of Contents

1 Introduction

1.1 L in brief

L is:

- A compiled language with a C-like syntax, and Lisp-like macros.
- It is an extensible programming language : even the syntax is modifiable at run-time.
- It is a mostly-safe language, with strong typing and encouraged confinement of dangerous constructs.
- Finally, thanks to extensibility, L is an *universal* language: it can be used both for low-level and system programming than for creating complex high-level applications. Using certain set of predefined modules, it could even be used as a scripting language.

So, L can be seen both as:

- C with stronger typing + extensible compiler support (macros, parser, expanders) + fully expression-based.
- or Lisp with low-level capabilities, static typing, support for custom syntaxes, and type-aware macros.

Hence the name, L: L combines C and Lisp.

But L is not just a combinaison of two languages: it a language of its own. Just like Lisp and C are both defined by a just a small number of specific characteristics, L's features make it unique.

1.2 L features

L has numerous features that makes it a cool language to hack with. Among them:

- L is a compiled language. It can thus run very fast. Compilation allows to track your errors sooner, and execution is faster than with other techniques. L programs can have very low memory footprint. L can sometimes outperfom C; for instance, its type system allows more agressive memory aliasing.
- L is interactive. Speed up your developpement by not using the "edit-compile-test" cycle developpement process. Write part of a function and immediately test it. Interactively compiling functions is very fast, so you don't have to wait for long compile/link cycles.
- L is extensible. If you need anything to add in the language you don't need to wait for 10 years that a committee decide to enhance the language. You just do it. Extensibility is at the heart of L programming, and is a programmation paradigm in itself.
- L is multi-paradigm. L's extensibility allows to write programs using different paradigm at the same time. I.e., you can freely mix functional, imperative, and object-oriented code.
- L has a flexible type system. Types are there to help you and automatically manage things for you; they don't get in your way.
- L helps you write safe programs. Traditionnal languages are split between too extremes:
  - Safe languages. Unfortunately, clever optimisations aren't allowed with them.

  - Unsafe language, like C. They allow to write any code, but programs written with them often crash or lead to security issues.

  With L, it is easy to write safe programs with some confined unsafe parts. Just write the unsafe parts carefully, and your program will be both optimized and won't crash.

  For instance, you can use pointers (both powerful and unsafe), but L provides many ways to avoid their use (for instance using Tuples, and multiple return values) or confine them (using macros, like the `foreach` macro for iterating on lists).

- L interfaces well with C. You can directly use C libraries from L, and use L libraries from C. No need to write long wrappers for that.
- L enables extreme factorisation of your programs. Typical L programs are a lot smaller than equivalent C programs. As such, they are easier to understand as a whole.

  As a corollary, with L, you can optimize your program without sacrificing readability. Dirty hacks can be confined and factorised. If you later change your mind about a hack, you just have to change one place, something that is not always possible e.g. in C.

2 Tutorial

2.1 L tutorial for the C programmer

In this first part, I will assume that you are already accustomed to a C like-language, preferably C. If you are not, skip to the unwritten node "L tutorial for the newbie programmer", or to "L tutorial for the Ruby programmer", or to "L tutorial for the Lisp programmer". Unfortunately for you, these tutorials are not yet written.

2.1.1 Translating C into L

First, you want to need at least as proficient with L than you were with C.

L is basically compatible with C, but there are some syntaxic differences.

Let's see our first example:

```
int
foo(int a, int b)
{
  return (a * (a + b);
}
```

In this short example, you can see that L syntax is quite close to the one in C. Still, here are the main differences that you will find when you want to translate C code into L :

2.1.1.1 Variables declaration

You have to precede all your local variable definitions with a let :

```
int
foo(int a, int b)
{
  let int c = a + b;
  return a * c;
}
```

I can already hear some old UNIX hackers already yelling : so L is more verbose than C, even more verbose than Java?

No: in L, type annotations are optional. That is, you can define exactly the same function if you use:

```
int
foo(int a, int b)
{
  let c = a + b;
  return a * c;
}
```

Type annotations are useful for the programmers for as checks of what is going on; you can see them as partial specifications of your program.

Passing the types of the parameters (and knowing the types of the different global variables) is in fact sufficient for deducing the types of all local variables; that's why type annotation is useless for the compiler.

Thus you avoid things like:

```
BufferedInputStream bufferedInputStream = new BufferInputStream(....);
```

which are quite redundant, and for which the type of the variable is obvious.

C's type qualifiers also have a different syntax: UNIMPLEMENTED

let Int : extern global_variable; let Float : static other_global_variable;

This ":" syntax will become clearer when we have seen Section 3.4.1 [Type classes], page 22.

2.1.1.2 Type declaration and utilisation

Type declaration (`typedef` in C) are written as follows:

```
type Toto = struct { Int foo; Int baz;};
```

L's convention for types is that they should be capitalized, and if you have an identifiers that spans over several words, separate them by underscores and capitalize each word, Like_That.

But this is is really just a convention; your own program can be written as you like.

The `type =` declaration isn't like `typedef`, because it introduces a new type that is incompatible with the first. (COMMENT : the = is maybe misleading in that case?)

If you write, for instance:

```
type Kelvin = Float;
```

then you cannot have:

```
let Kelvin k = 3.0;
```

You have to do this instead: (UNIMPLEMENTED: does not work for now, explicit casts needed)

```
let Kelvin k = Kelvin(3.0);
```

or simply:

```
let k = Kelvin(3.0);
```

Similarly, the `+` operations work on `Int`s, and because `Kelvin`s are not `Int`s, you cannot add `Kelvin`s.

L provides a shortcut (UNIMPLEMENTED) for specifying which operations are still allowed for the new type:

```
type Kelvin = Float | allows +,-;
```

Forbiding `*` is interesting for instance, because multiplying Kelvins between them make no sense. Multiplying them by a Float makes sense, however. (UNIMPLEMENTED: so we need a way to tell that).

Finally, writing all this can be factorized to:

```
import std.unit;

type Kelvin = Float : unit;
```

`unit` declares a type that does all this (in fact, that respects the `unit` interface). `numeric` declares a type that has `+`, `-`, `*`, `/`. All these are in fact type classes, see Section 3.4.1 [Type classes], page 22.

If you just want something similar to C `typedef`, you can use L's `typealias`:

```
typealias Floating_Point = Float;
typealias Integer = Int;
```

defines Integer as an alias for Int. (i.e. they are the same type with just different names.)

2.1.1.3 Grammar for types

The grammar for types is quite different from C's one. First, all types information are grouped together; you don't have some that are prefix to the identifier, and some postfix (like `*` and `[]` are in C).

Second, the notation for functions is simply `<-`.

Here are some examples:

C	**L**
int i;	let Int i;
int *pi;	let Int *pi;
int pi[3];	let Int[3] pi;
int *pi[3];	let Int *[3] pi;
int (*)pi[3];	let Int [3]* pi;
float (*fi)(int);	let (Float <- Int)* fi;

2.1.1.4 Type construction and memory allocation

By default when creating a new type, L creates also a new generic constructor, which depends on the created type.

For instance, you must use

```
type Kelvin = int;

let degree = Kelvin(24);
```

to create new Kelvin objects.

When the type is a struct, or a pointer to a struct, the constructor is used as in the following:

```
type Point = struct { int x; int y; } *;
type Point_Struct = struct { int x; int y; };

...

let p = Point(x:4, y:5);
let p2 = Point_Struct(x:4, y:5);
```

The difference between the two is that `p2` is allocated on the stack (as a regular C struct), while `p` is allocated on the heap.

The `x:` and `y:` are keys for keyword arguments. The rationale behing using them is that if, later, you change your mind about the structure contents, you will know it immediatly (this wouldn't be the case if you used normal arguments).

In order not to forget this '*', you can use the `record` type constructor:

```
type Point = record { int x; int y; };
// same as type Point = struct { int x; int y; } *;
```

In fact, it isn't really the same: `record`-declared types are by default garbage collected, whereas `struct *`-declared types must be explicitly freed.

This can be overriddent by the `alloc:` parameter:

```
let p1 = Point(x:4, y:5)
// Allocated and managed by the most appropriate garbage collector

let p2 = Point(x:4, y:5, alloc:gc) // Same as p1

let p3 = Point(x:4, y:5, alloc:refcount)
// Same as p1, but explicitly asks for the refcount GC

let p4 = Point(x:4, y:5, alloc:mark_sweep)
// Same as p1, but explicitly asks for the mark&sweep GC

let p5 = Point(x:4, y:5, alloc:heap) //No garbage collection is done

let p6 = Point(x:4, y:5, alloc:stack) //Allocated on the stack.
```

By default, all the type fields have to be given as arguments.

2.1.2 Extensions to C

L has numerous extensions to C, but here are some extensions that are useful when programming in the small

2.1.2.1 Blocks and conditional as expressions

In L, blocks can return a value. For instance:

```
let Int a = { let Int sum = 0;
              let Int i = 0;
              for(i = 0; i <= 10; i++) { sum += i; }
              sum };
```

The above code creates a new block where `sum` and `i` are two variables created in the block. An iteration is done, and then the value of sum is returned, and affected to `a`.

Note the syntax : if a block is supposed to return a value, it is composed of a list of *statements* followed by one expression. By contrast, the block in the `for` does not return a value, and is composed only of `statements`. This will be detailed more in Section 3.2.1 [Extending the syntax], page 18.

This will be very useful when you write Section 2.1.2.5 [Macros], page 9; but it is also good programming style : it is better to pass values around using this mechanism than creating a new variable to do it.

Conditionals can also return a value. Here is how to compute the absolute value of a :

```
let Int abs_x = if(x >= 0) x else -x;
```

This eliminates the need for a distinct `? ... :` operator.

Note: You can still use conditional as you would in C. The above example could also be written:

```
let Int abs_x;
if(x >= 0)
   abs_x = x;
else
   abs_x = -x;
```

or even:

```
let Int abs_x = x;
if(x < 0)
  abs_x = -x;
```

2.1.2.2 Tuples

Tuples are a way to manipulate several values at once.

Creating a new tuple does not allocate memory. Tuples don't have addresses, their content is not placed contiguously in memory. They *really* are just a way to consider several values at once.

For instance, the tuple (4, 5, 'toto') is a constant. Thus you can create multiple-values constants with L.

When a tuple contain *complex* expressions (that is to say, anything except a constant or a variable), *the order of evaluation is defined to be from the left to the right.*

- Passing arguments to functions is done using a tuple. I.e.

  ```
  let h = hypot(side1, side2);
  ```

  calls `hypot` with the tuple `(side1, side2)`. As a side effect of the order of evaluation rule stated before, *function arguments are evaluated from left to right.* In C, this is undefined, which causes many portability problems.

- Return values of functions are also tuples. So, functions can return multiple values in L:

  ```
  let (return_value, is_present) = gethash('key');
  let (num_char_scanned, num1_scanned, num2_scanned) = scanf("%d %d");
  ```

  In most cases, the use of pointers to return several data should be replaced by a multiple return value form. This is one of the example where pointers can be easily avoided in L (thus providing more safety).

  In the implementation, the return values are passed in clobbered registers, and is very efficient.

- Finally, it is also useful to do multiple affectations simultaneously, without having to explicitly declare local storage:

  ```
  (str,len) = ("foo",4); // same as str = "foo"; len = 4;

  (point.x, point.y) = (3.0, 5.0);

  a = 2;
  b = 3;

  (a,b) = (b, a); // Now b = 2, and a = 3.
  ```

```
let (c,d) = (4, 'foo');
```

When doing multiple affectations, you can "skip" one by using the special symbol '_':

```
let i = 0;
(a,_,c) = (++i,++i,++i); //a = 1; c=3;
```

This is most useful when you want to receive values from functions:

```
(value, present) = gethash(key, hash_table);
(value, _) = gethash(key, hash_table);
value = gethash(key, hash_table);
```

Finally, on an implementation note, using tuple is higly efficient, because each component of a tuple can be a register.

For instance, the `(a,b) = (b,a)` construct may use the efficient `xchg` instruction on CISC machines; It is difficult for a standard C compiler to use these instructions, and the corresponding code would use three instructions and a supplementary register.

Tuple is thus an both a pleasant and efficient abstraction.

Note: Depending on the architecture, Word64 can be or not a tuple. But most code can ignore this fact and completly ignore the issue.

2.1.2.3 Keyword and default arguments

L functions can have default arguments UNIMPLEMENTED, like C++ ones:

```
Int foo(Int bar = 3)
{
  bar
}

foo(5); //5
foo();  //3
```

L functions argument passing may also be done using keyword arguments UNIMPLEMENTED:

```
Int foo(Int bar, Int baz)
{
  10 * bar + baz
}
```

If you write:

```
Int foo(Int bar = 1, Int baz)
{
  10 * bar + baz
}
```

Then `foo` can only be called like this:

```
foo(3, 4)  //OK, 34
foo(baz:5) //OK, 15
foo(5)     //Wrong
```

Use of keyword arguments is a really good style when you create functions that create complex objects (structures and records), especially when they contain fields that have the same type.

For instance:

```
let richards_shoes = Shoes(color:Green, sole_color:Brown)
```

Is much better style than

```
let richards_shoes = Shoes(Green, Brown)
```

In general, it is better to use them when a function has several arguments of the same type, or several arguments at all. It makes your code more readable.

You can also use them for "cosmetic" usage, as in:

```
foreach(a, in:list) {...}
foreach(element:a, in:list) {...}
divide(25, by:73);
```

2.1.2.4 Recursive types and functions

L handle recursive types and functions very well. In functional programming languages, recursive type definition is often "work-arounded" by using a "rec" keyword.

In L, the rule is that "every code simultaneously entered is treated as a whole".

For instance, if you feed L with:

```
type Toto = struct { Tata; } *;
type Tata = struct { Toto; } *;
```

L would correctly interpret this. UNIMPLEMENTED: not yet. Only

```
type Int_List ; struct { Int head; Int_List tail; }*
```

recursive definition works for now.

But if you feed L with:

```
type Toto = struct { Tata; } *;
```

and (after) with:

```
type Tata = struct { Toto; } *;
```

You would get an error after the first sentence, because the definitions you supply is incomplete.

See Section 3.3.3 [Chunk], page 22 for more informations.

2.1.2.5 Macros

Macros are L's replacement for the C preprocessor, or C++ templates. It allows you to factorize patterns your code, so that your code is clearer and easier to understand.

More specifically, macros are a templating language that replaces a construct by a sequence of code.

Introduction

Let's start with an example to see how it works:

```
macro square(x)
{ let _value = $x;
  _value * _value }
```

The effect of the macro is as follow: if you write:

```
let Int number = square(4);
```

It will be converted into:

```
let Int number = {let _value = 4; _value * _value; };
```

thus yielding 16.

If you have typed:

```
let Float number = square(0.5);
```

it would have been converted to

```
let Float number = {let _value = 0.5; _value * _value; };
```

thus yielding 0.25.

The process of converting a "macro call" into what it stands for is called *macro expansion*.

Here you can see how macros are type independent, since you can use them with Floats and Ints. In fact, you can use them on every type T for which the operation `*(T,T)` exists.

Here you can see the interest of:

- Let without type annotation (that helps creating type independent macros easily), and
- Blocks that can return expressions (it allows to create local variables and still return a value).

Macros and type

One of the distinguished features of L macros is that they are typed. L macros do not just perform text rewriting, like the C processor : they can check type, and even act differently according to the type.

```
macro power(Float x, Float y)
{
  powf($x, $y)
}

macro power(Long x, Long y)
{
  powl($x, y)
}
```

Note: this is in fact the way how overloading is implemented. UNIMPLEMENTED: or will be.

The use of types allows earlier type-checking. If we take the previous example:

```
macro square(x)
{ let _value = $x;
  _value * _value }
```

And if we write `square("25")`, we will end up with:

```
error: (in expansion from square)
     :  in file test.c, line 25:
       * does not accept (String, String) arguments.
```

If we had written instead:

```
macro square(Int x)
{ let _value = $x;
  _value * _value }
```

The error message would have become:

```
error: in file test.c, line 25: square does not accept String argument.
```

which is much more readable.

But if we want the definition of square to be more generic (for instance, to allow both Int and Floats, and any type T that have a `*(T,T)` operation), just specify the macro like this:

```
macro square(x : numeric)
{ let _value = $x;
  _value * _value }
```

UNIMPLEMENTED: this relies on type classes, that are not yet implemented.

Use of macros

The main use of macros is not for `square`-like examples : inline functions are here for this (even if inline functions are implemented as macros in L).

It is mainly useful to introduce new programming constructs. For instance, in L, `while` is not part of the language: it is defined as a macro. Here is its definition:

```
macro while(Bool condition, body)
{
  loop {
    if(!$condition)
      break;
    $body; }
}
```

Then `while` can be used like this:

```
let i = 25;
while(i > 0,
    { print(i--, '\n');
    });
```

This isn't really like the C while. To really obtain the C definition in this case, you also have to hook the parser: this is explained in Section 3.2.1 [Extending the syntax], page 18.

`++` and `--` are also not part of the language, but defined as macros, and could be defined as:

```
//pre_inc(x) is the same as ++x
macro pre_inc(x)
{ x = x+1 }

//post_inc(x) is the same as x++
macro post_inc(x)
{ let _temp = x;
  x = x + 1;
  _temp }
```

Note: In fact, to make the Cocytus more readable, and the C output look more like real code, `while`, `++` and `--` are part of the Cocytus language. But their default implementations are the above ones.

Care with macros

Use macros with care: for instance if you define

```
macro square(Int x)
{$x * $x}
```

Then `square(i++)` will be transformed into `(i++) * (i++)` which is certainly not what you want.

In this example, you must write:

```
macro square(Int x)
{ let _x_value = $x;
  _x_value * _x_value }
```

That would yield the correct result. (Note: in this case, an inline function would have been better than a macro).

In general, it is dangerous to insert an argument directly more than once, and you must do it only if it is the effect that you want to achieve, as in

```
macro do_twice(x)
{ $x;
   $x;
}
```

The second problem you can encounter is called *variable capture*. For instance, if you define `do_times` like this: (the following example is inspired by Paul Graham's On Lisp):

```
macro do_times(Unsigned_Int max, body)
{ let max_value = $max;
  let i = 0;
  loop {
  if(i++ == max_value)
     break;
   $body;
   }
}
```

then your code would work in most cases:

```
do_times(4) { print("hello\n"); } //print hello four times
```

But if you write:

```
let i = 0;
do_times(4) { i++; }
//Here, i is still 0
```

(Note: if you had written `do_times(5)` instead of `do_times(4)`, then the code would never have returned.)

The reason is that the local variable `i` used by the `do_times` macro and the one used in the block passed as an argument “clash” because they have the same name. Here is the full expansion of the above call:

```
let i = 0;
{ let max_value = 4;
  let i = 0;
  loop {
  if(i++ == max_value)
     break;
   { i++ };
  }
}
```

The naive solution to this problem is to use such ugly names that they can never clash with identifiers that a programmer would choose, like `srchroukjuk239`. This doesn't really make your code pretty, and worse, it does not work when macro definitions are nested:

```
do_times(4) { do_times(5) { print("Hello\n");}}
```

The real solution to this problem is to use *generated symbols*, i.e. to generate a new symbol for the name of the variable for each new expansion, that are guaranteed to be uniques, and thus never clash.

L provides a simple way to do this: just prepend a '_' to your identifiers in your template code. L will recognize that and instead will generate a new symbol for each new expansion of the macro. (Note: more over, the generated symbols are generated in such a way that they are readable, and that you can find out in what macro they have been generated. This simplifies debugging of macros).

In normal code, you do not have the right to begin an identifier by an underscore (as it is forbidden in C).

Thus, the correct definition for the above macro is:

```
macro do_times(Unsigned_Int max, body)
{ let _max_value = $max;
  let _i = 0;
  loop {
  if(_i++ == _max_value)
     break;
   $body;
   }
}
```

And macro expansion becomes:

```
let i = 0;
{ let do_times#max_value_1 = 4;
  let do_times#i_1 = 0;
  loop {
  if(do_times#i_1++ == do_times#max_value_1)
     break;
   { i++ };
  }
}
```

Note: You may wonder why all `let` definitions are not simply transformed into generated symbols. First, sometimes you may want to generate a new symbol, even if there is no `let` (imagine, for instance, that you have written a `my_let` macro that expands into `let`).

Second, there are some cases where variable capture is actually wanted, and it would be silly to prevent it.

L can however generate a warning when it detects a variable capture because of a `let`; this warning can be desactivated like this:

```
macro with_log_level(number, body) captures(log_level)
{
  let log_level = number;
  $body;
}
```

although this changes `log_level` only in the current lexical scope; compare it with that:

```
macro with_log_level(number, body)
{
 let _save_log_level = level;
 log_level = number;
 $body
 log_level = _save_log_level;
}
```

Using multiple levels of macros

Use of macros can be nested, as in:

```
let x = 0;
while(i <= 25) { i = square(x); print(i); x++; }
```

In this case, inside expansions occurs *before* outside expansions. That is to say, the above example is first converted into:

```
let x = 0;
while(i <= 25) { i = {let square#temp = x; square#temp * square#temp }
```

Before being converted to:

```
let x = 0;
loop {
   if(!(i <= 25)) break;
   {
```

```
        i = {let square#temp = x; square#temp * square#temp }
    }
}
```

Conclusion

Macros are a templating language for L; it allows code to be more understandable, and to let the compiler do things for you.

The combination of macros and types allows many powerful definitions, that depends on the type of the arguments. You can really define new programming language constructs, and use them as if they were part of the language.

Note: as you have to use `{ }` in a macro, variables declared in a macro are always local to a macro, and are thus limited to computation inside the macro. This catch programmers errors that are hard to find.

If you want to implicitly declare new variables in your code, you have to use the more powerful Section 2.1.2.12 [Expanders], page 15. In general, if you want to generate code and that a templating language is not powerful for this, you have to use the more general Section 2.1.2.12 [Expanders], page 15.

Note: using macros create many redundant code that a programmer wouldn't have written. Fortunately, this redundance is often eliminated when converted to SSA Form:

```
let a = { let temp i; i = 3; i };
```

is converted in :

```
let a = 3;
```

in the optimisations pass of the compiler.

2.1.2.12 Expanders

Expanders are a generalisation of the notion of macro. (In fact, it is more accurate to say that macros are a special case of expanders).

Instead of using a fixed template for replacing code, code can be dynamically constructed based on the the parameters given (which do not need to be expanded).

TO_WRITE

2.1.2.13 Extensible syntax

TO_WRITE: how to hook the parser, the parse language...

2.1.3 Restrictions to C

TO_WRITE

3 L presentation

The L compiler is composed of many parts.

TO_COMPLETE

3.1 L structure

L' compilation process is composed of several parts:

- The '`Parser`', that takes the buffer of characters and converts it into a tree of forms, the abstract syntax tree. The input of the parser is what is called "L"; and the parser acts according to the Section 3.2 [L concrete syntax], page 17.
- The '`Expander`', transforms the abstract, high level forms into concrete, low level ones. It is responsible for macro expansion. It is also called '`Malebolge compiler`', because the parsed L forms are called Malebolge. Malebolge deals with the abstract syntax of the language.
- The '`Code generators`', take the low level forms and transform it into output. It is highly dependent of the output; we could consider that there are in fact several code generators, and these are called '`backends`'. It is also called '`Cocytus compiler`', because the low level forms are the Cocytus language. Cocytus deals with the precise semantics of the language.

Malebolge is a bit to Cocytus what C++ is to C: a super-set of the language, that allows more abstraction, more hiding of informations. But reading C++ code, one cannot exactly says what will be executed; by contrast any experienced C programmer knows exactly what's going on when he reads a C program. This is exactly the same thing for Malebolge and Cocytus: reading a Malebolge program provides a global understanding of the program, you can read the intention of the programmer, understand the algorithms. Reading the Cocytus transformation, you can read how this intention was transformed by the Malebolge compiler, and how it is translated into machine language.

Malebolge programs can be converted to Cocytus : both languages coexists. This is important, because it allows a programmer to manually check what's going on in its program, AND to have precise control over the language. So in L, we combine both advantages. Separating the language into two levels renders easy both the global comprehension and deep comprehension of a program.

We already have a complete compilation process that takes L source code, parses it, expands it, and generates x86 code dynamically in memory (that's why the compiler is *interactive*).

The following (outdated) diagrams shows how the complete L implementation could look like:

```
                      L source file (.l)
                             |
                          parser* (produce L forms)
                          /  | \
       parsed file generator  |  sexped file generator
                        /    |    \
```

```
      L parsed file (.pl) |    L sexped file (.sl)
                          |
                          |
                    Malebolge compiler*  (produce L Cocytus forms)
                      /   |     \     \------------------\
L cocytus source generator |    L slim binaries genetor    \
                    /     |       \                        GNU C generator
                   /      |      L binary file (.bl)           \
 L cocytus source (.cl)    |                                   C output file (.c)
                          |
                     Cocytus compiler* (calls the backend functions)
                   /      |         \
                  /       |          \-------\-----------------
                 /   dynamic x86 code*        \                \
     L GCC frontend                       ANSI C generator      \
               /                                            L header files (.lh)
             GCC
             /  \---------------
            /                   \
        assembly file (.S)   object file (.o)
```

*: already implemented

3.2 L concrete syntax

The parser handle the concrete syntax of the language.

TO_WRITE

S-expressions (sexps): a notation to represent trees.

Concept of tree : `a + b * c` is converted into `(* (+ a b) c)`. In this representation, the first symbol represents the head of the tree, and subsequents fields represent the sons, that can be trees themselves. So the above code is really:

```
*--- + --- a
 \    \----b
  \--c
```

Another example: L's code

```
function1(toto, function2(24,49), 3+i)
```

has the tree representation:

```
                function1
               /     |    \
           toto function2  +
                  /   \    |\
                 24   49   3 i
```

And the sexp representation:

```
(function1 toto
           (function2 24 49)
```

```
              (+ 3 i))
```

So the sexp notation is just a convenient representation to represent trees.

3.2.1 Extending the syntax

TO_WRITE

3.3 Cocytus

The Cocytus is the set of core language constructs of L. Every L code is transformed into an assembly of Cocytus operations, that form a tree.

The goal of the Cocytus language is to have a language that is semantically non ambiguous; that is to say, that can express very precisely what the computer will do. Cocytus output should be readable; it can be used to manually verify what a code does. It is much higher level than assembly, approximatively of equal expressiveness than C, but more precise, less ambiguous.

3.3.1 Language constructs

In L, new language constructs can be defined, and existing language constructs can be redefined. There are many cases where this may be useful:

- If you have a processor with special instructions, you need to create some backend-specific language constructs to exploit them.
- In general, every addition to the language that need some assembly manipulations require a new language construct.

L defines a few language constructs that, taken alone, make L approximatively as expressive as C. To have full expressiveness, you need macros and expanders, that are defined in the next section Section 3.4 [Malebolge], page 22.

The standard language constructs are the following:

3.3.1.1 Constructs for local structure

`seq` (*form1..., form_n*) [Language construct]

`seq` Executes each of its subforms in turn, and returns the result of the last one. `seq` must have at least one form.

L's standard syntax for this construct is `,`:

```
x = 3, y = 4, x * y
```

has for abstract syntax tree:

```
(seq (= x 3)
     (= y 4)
     (*#Int x y))
```

that has for result `12`.

`block` (*form1..., form_n*) [Language construct]

`block` acts like `seq`, except that it also begins a new block of code. All subsequent `let` definitions have a scope that ends at the end of the current block. Block must have at least one form, like seq.

Note that L's blocks can return a value, unlike C ones. The syntax for L blocks thus differs from C's (even GNU C's) one:

```
let Int a = { let Int sum = 0;
              for(let Int i = 0; i <= 10; i++)
                { sum += i; }
              sum };
```

This code creates two local variables to do a local calculation, before returning a result. This is particularly handy in combination with macros.

`let` (*type_form, variable_name*) [Language construct]

`let` creates a new local variable that exists from its declaration until the end of the block.

A `let` form can be used as a lvalue, like in the construct:

```
let Int i = 3;
```

that has for abstract syntax tree:

```
(= (let Int i) 3)
```

but it cannot be used as a rvalue.

3.3.1.2 Constructs for global structure

`define` (*definition_type name rest*) [Language construct]

`define` defines NAME as being a DEFINITION_TYPE with value REST. This special form just really calls the definer associated with DEFINITION_TYPE, with parameters DEFINITION_TYPE, NAME, and REST.

Exemple of DEFINITION_TYPE are `function` (for defining new functions) `type` (for creating new types), `expander`, `thread-local` (unimplemented), `global` for thread-local and global variables.

See Section 3.3.2 [Definers], page 21 for the details.

3.3.1.3 Constructs for changing flow of control

`goto` (*label_name*) *UNIMPLEMENTED* [Language construct]

`goto` branches to the label LABEL_NAME. For the goto to be valid, the following condition must be met:

The label must appear in a scope "accessible" by the goto instruction, i.e. either the current scope or any parent enclosing scope. It is an error to jump to a label to an unreachable scope.

`return` (*value*) *UNIMPLEMENTED* [Language construct]

Aborts the execution of the current function and returns to the caller, with return value VALUE.

3.3.1.4 Constructs for iteration

You will notice that L does not have any of the standard constructs for iteration built-in, like C `for`, `while`, or `do ... while`. These can be defined by userlibraries; so a standard library defines them, but they are not part of the language.

`loop` (*form*) [Language construct]

Repeatly execute forms, until it reaches one of `break`, `continue`, or one of the two preceding change of flow of control commands `goto` (if the label is outside of the loop) or `return`.

`break` () [Language construct]

`break` exits the current loop; i.e., acts as if the enclosing loop was executed

`continue` () [Language construct]

`continue` continues the execution at the beginning of the loop.

3.3.1.5 Construct for affectation

L can be close to hardware, and thus is an imperative language, in the sense that it defines an imperative construct, `=`. By restricting its use, you can obtain a fully functional language if you prefer; this is discussed in further sections.

`=` (*assignee expression*) [Language construct]

`=` affects the value of EXPRESSION to ASSIGNEE, that must be a correct lvalue.

3.3.1.6 Constructs for pointer manipulation

L permits the use of pointers; not doing so result in a huge performance penalty as seen in the "modern" languages, and give up all hope to do low level programming. L is so an unsafe language.

However, L make it easy to hide the use of pointers behing safe constructs; if you make the usage of these constructs mandatory (which is not feasible by now), you transform L into a safe, AND efficient, language.

`ref` [Language construct]

`deref` [Language construct]

`deref` can be used as an lvalue

3.3.1.7 Constructs for structure manipulation

3.3.1.8 Construct for aggregating several values together

`tuple` *expression_1 ... expression_n* [Language construct]

L has a builtin notion of tuple, that is quite different from what is called tuple in many different languages.

L's tuple only purpose is to consider several values at once. In particular, *no special assumption is made about the location of the components of the tuple.* Unlike the structure, tuple components are not placed contiguously in memory, for instance.

The following code:

```
(block (let Int a)
       (let Int b)
       (tuple a b))
```

does not do anything, for instance.

Tuples are really useful when it comes to *simultaneous affectations*. For instance:

```
(a,b) = (b,a);
```

exchange the values in `a` and `b`. As there is no assumption behind the memory placement of the tuple, potentially optimized instructions can be used here; for instance if `a` and `b` where stored in registers, the above example could have used the x86 instruction `xchg`, which is never used by a normal C compiler.

L defines the order of evaluation of the expressions in a tuple to be from left to right. Unlike C programs, L programs can rely on this fact.

It is also important to notice that all expressions of the tuple are evaluated before the assignment takes place. This is what makes the above example to work; as opposed to sequential affectation.

3.3.1.9 Construct for function calling

funcall: calls a function. Takes a tuple as an argument, returns a tuple: thus we have multiple return values (this alleviates many use of pointers).

UNIMPLEMENTED: partial affectation of a tuple, using _:

```
(x,_,color) = f(i);  f: (Float,optional Float, Color)<-(Int);
(x,y,_) = f(i);   f: (Float,Float,optional Color)<-(Int);
(x,y) = f(i);   f: (Float,Float,optional Color)<-(Int);
x = f(i);   f: (Float,optional Float,optional Color)<-(Int);
```

3.3.2 Definers

Definers are used to define new global values. Standard L cocytus package provides several standard definers:

`function` [Definer type]

This definer makes it possible to define new functions, that can later be compiled.

`type` [Definer type]

With this definer, you can create new types

L's style encourage creation of new definer. For instance, imagine that you are developping a PCI device driver interface. Then, a device driver should look like:

```
pci_device_driver my_driver
{
  name: "my-driver";
  probe: my_probe_function;
  remove: my_remove_function;
  suspend: my_suspend_function;
  resume: my_resume_function;
}

Error_Code my_probe_function(Pci_Info inf)
{ ... }
```

Any change of interface would be immediately known at compile time: for instance, if a suspend is deprecated, the definer can warn the developper when it is called; and so on.

UNIMPLEMENTED This is also how Cocytus backends should be declared:

```
Cocytus_Backend pretty_GNU_C_backend
{
  let: pgc_compile_let;
  =: pgc_compile_assign;

}
```

A warning or error message could then be issued when a Cocytus backend does not follow a change in Cocytus; or does not fully implement it for instance.

Finally, the ability to add new definers make it possible to transform L into a declarative language, or more generally use declarative constructs, so writing what you want to have instead of writing how to obtain it.

3.3.3 Chunk

Cocytus is compiled by chunks. If you reference something, it has to be defined in the same chunk, or already defined in a previous chunk.

3.4 Malebolge

TO_WRITE

3.4.1 Type classes

3.4.2 Defining new macros

3.4.3 Creating new expanders

3.4.4 Defining coercions