L2

L2 is an attempt to find the smallest most distilled programming language equivalent to C. The goal is to turn as much of C's control structures, statements, literals, data structure constructs, and functions requiring compiler assistance (setjmp, longjmp, assume, ...) into things definable inside L2 (with perhaps a little assembly). The language does not surject to all of C, its most glaring omission being that of a type-system. However, I hope the examples below will convince you that the result is still pretty interesting. And if that is not enough, I recommend that you take a look at the implementation of a self-hosting compiler for L2 that accompanies this project and compare it to the compiler for bootstrapping it written in C.

The approach taken to achieve this has been to make C's features more composable, more multipurpose, and, at least on one occasion, add a new feature so that a whole group of distinct features could be dropped. In particular, the most striking changes are that C's:

1. irregular syntax is replaced by S-expressions; because simple syntax composes well with a non-trivial preprocessor (and no, I have not merely transplanted Common Lisp's macros into C)
2. loop constructs are replaced with what I could only describe as a more structured variant of setjmp and longjmp without stack destruction (and no, there is no performance overhead associated with this)

There are 9 language primitives and for each one of them I describe their syntax, what exactly they do in English, the i386 assembly they translate into, and an example usage of them. Following this comes a listing of L2's syntactic sugar. Then comes a brief description of L2's internal representation and the 6 functions that manipulate it. After that comes a description of how a meta-expression is compiled. The above descriptions take about 8 pages and are essentially a complete description of L2. Then at the end there is a list of reductions that shows how some of C's constructs can be defined in terms of L2. Here, I have also demonstrated closures to hint at how more exotic things like coroutines and generators are possible using L2's continuations.

Contents

Getting Started

Building L2

```
./build_bootstrap
./build_selfhost
```

In this project there are two implementations of L2 compilers. One implementation is the bootstrap compiler that comprises 3700 lines of C code which compiles in under a third of a second. The other implementation is a self-hosting compiler written in about 3700 lines of L2 code (the meta-program accounts for about 1100 lines and the program accounts for the other 2500 lines) which compiles to a roughly a 700KB executable in under 4 seconds. Both of them produce identical object code (modulo padding bytes in the ELFs) when given identical inputs. **The bootstrap compiler needs a Linux distribution running on the x86-64 architecture with the GNU C compiler installed to be compiled successfully.** To build the bootstrap compiler, simply run the `build_bootstrap` script at the root of the repository. This will create a directory called `bin` containing the file `l2compile`. `l2compile` is a compiler of L2 code and its interface is described in the next section. To build the self-hosting compiler, simply run the `build_selfhost` script at the root of the repository. This will replace `l2compile` with a new compiler that has the same command line interface.

The Compiler

```
./bin/l2compile (metaprogram.o | metaprogram.l2) ... - program.l2 ...
```

L2 projects are composed of two parts: the program and the metaprogram. The program is the end product; the stuff that you want in the output binaries. The metaprogram is the code that the compiler delegates to during the preprocessing of the program code. The L2 compiler begins by loading the metaprogram into memory. For the parts of the metaprogram that are object files, the loading is straightforward. For the parts of the metaprogram that are L2 files, they cannot simply be compiled and loaded as they may also need to be preprocessed. Hence a lazy compilation scheme is implemented where an object file exposing the same global symbols as the L2 file is loaded, and only later on when one of its functions is actually called will the compilation of the corresponding L2 code actually be done. The important gain to doing this is that the aforementioned compilation now happens in the environment of the entire metaprogram, that is, the metaprogram can use its entire self to preprocess itself. Once the metaprogram is loaded, its parts are linked together and to the compiler's interface for metaprogramming. And finally each part of the program is compiled into an object file with the assistance of the metaprogram.

Example

file1.l2

```
(function foo (frag buf) [@fst frag])
```

file2.l2

```
(function bar ()
  [putchar (literal 0...01100011)])
(foo [putchar (literal 0...01100110)])
[putchar (literal 0...01100100)]
```

Running `./bin/l2compile "./bin/x86_64.o" file1.l2 - file2.l2` should produce an object file file2.o. file2.o when called should invoke the function `putchar` with the ASCII character 'f' and then it should invoke the function `putchar` with the ASCII character 'd'. And if its function `bar` should be called, then it will call the function `putchar` with 'c'. Why is it that the first invocations happen? Because object code resulting from L2 sources are executed from top to bottom when they are called and because the expression `(foo [putchar (literal 0...01100110)])` turned into `[putchar (literal 0...01100110)]`. Why is it that the aforementioned transformation happened? Because `(foo [putchar (literal 0...01100110)])` is a meta-expression and by the definition of the language causes the function `foo` in the metaprogram to be called with the fragment `([putchar (literal 0...01100110)])` as an argument and the thing which `foo`

then did was to return the first element of this fragment, `[putchar (literal 0...01100110)]`, which then replaced the original `(foo [putchar (literal 0...01100110)])`.

Expressions

Begin

```
(begin expression1 expression2 ... expressionN)
```

Evaluates its subexpressions sequentially from left to right. That is, it evaluates `expression1`, then `expression2`, and so on, ending with the execution of `expressionN`. Specifying zero subexpressions is valid. The return value is unspecified.

This expression is implemented by emitting the instructions for `expression1`, then emitting the instructions for `expression2` immediately afterwords and so on, ending with the emission of `expressionN`.

Say the expression `[foo]` prints the text "foo" to standard output and the expression `[bar]` prints the text "bar" to standard output. Then `(begin [foo] [bar] [foo] [foo] [foo])` prints the text "foobarfoofoofoo" to standard output.

Literal

```
(literal b63b62...b0)
```

The resulting value is the 64 bit number specified in binary inside the brackets. Specifying less than or more than 64 bits is an error. Useful for implementing character and string literals, and numbers in other bases.

This expression is implemented by emitting an instruction to `mov` an immediate value into a memory location designated by the surrounding expression.

Say the expression `[putchar x]` prints the character `x`. Then `[putchar (literal 0...01100001)]` prints the text "a" to standard output.

Storage

```
(storage storage0 expression1 expression2 ... expressionN)
```

If this expression occurs inside a function, then space enough for `N` contiguous values has already been reserved in its stack frame. If it is occuring outside a function, then static memory instead has been reserved. `storage0` is a reference to the beginning of this space. This expression evaluates each of its sub-expressions in an environment containing `storage0` and stores the resulting values in contiguous locations of memory beginning at `storage0` in the same order as they were specified. The resulting value of this expression is `storage0`.

`N` contiguous words must be reserved in the current function's stack-frame plan. The expression is implemented by first emitting the instructions for any of the subexpressions with the location of the resulting value fixed to the corresponding reserved word. The same is done with the remaining expressions repeatedly until the instructions for all the subexpressions have been emitted. And then second emitting an instruction to `lea` of the beginning of the contiguous words into a memory location designated by the surrounding expression.

The expression `[putchar [get (storage _ (literal 0...01100001))]]`, for example, prints the text "a" to standard output.

If

```
(if expression0 expression1 expression2)
```

If `expression0` is non-zero, then only `expression1` is evaluated and its resulting value becomes that of the whole expression. If `expression0` is zero, then only `expression2` is evaluated and its resulting value becomes that of the whole expression.

This expression is implemented by first emitting an instruction to `or expression0` with itself. Then an instruction to `je` to `expression2`'s label is emitted. Then the instructions for `expression1` are emitted with the location of the resulting value fixed to the same memory address designated for the resulting value of the `if` expression. Then an instruction is emitted to `jmp` to the end of all the instructions that are emitted for this `if` expression. Then the label for `expression2` is emitted. Then the instructions for `expression2` are emitted with the location of the resulting value fixed to the same memory address designated for the resulting value of the `if` expression.

The expression `[putchar (if (literal 0...0) (literal 0...01100001) (literal 0...01100010))]` prints the text "b" to standard output.

Function

```
(function function0 (param1 param2 ... paramN) expression0)
```

Makes a function to be invoked with exactly `N` arguments. When the function is invoked, `expression0` is evaluated in an environment where `function0` is a reference to the function itself and `param1`, `param2`, up to `paramN` are the resulting values of evaluating the corresponding arguments in the invoke expression invoking this function. Once the evaluation is complete, control flow returns to the invoke expression and the invoke expression's resulting value is the resulting value of evaluating `expression0`. The resulting value of this function expression is a reference to the function.

This expression is implemented by first emitting an instruction to `mov` the address `function0` (a label to be emitted later) into the memory location designated by the surrounding expression. Then an instruction is emitted to `jmp` to the end of all the instructions that are emitted for this function. Then the label named `function0` is emitted. Then instructions to `push` each callee-saved register onto the stack are emitted. Then an instruction to push the frame-pointer onto the stack is emitted. Then an instruction to move the value of the stack-pointer into the frame-pointer is emitted. Then an instruction to `sub` from the stack-pointer the amount of words reserved on this function's stack-frame is emitted. After this the instructions for `expression0` are emitted with the location of the resulting value fixed to a word within the stack-pointer's drop. After this an instruction is emitted to `mov` the word from this location into the register `eax`. And finally, instructions are emitted to `leave` the current function's stack-frame, `pop` the callee-save registers, and `ret` to the address of the caller.

The expression `[putchar [(function my- (a b) [- b a]) (literal 0...01) (literal 0...01100011)]]` prints the text "b" to standard output.

Invoke

```
(invoke function0 expression1 expression2 ... expressionN)
[function0 expression1 expression2 ... expressionN]
```

Both the above expressions are equivalent. Evaluates `function0`, `expression1`, `expression2`, up to `expressionN` in an unspecified order and then invokes `function0`, a reference to a function, providing it with the resulting values of evaluating `expression1` up to `expressionN`, in order. The resulting value of this expression is determined by the function being invoked.

`N+1` words must be reserved in the current function's stack-frame plan. The expression is implemented by emitting the instructions for any of the subexpressions with the location of the resulting value fixed to the corresponding reserved word. The same is done with the remaining expressions repeatedly until the instructions for all the subexpressions have been emitted. Then an instruction to `push` the last reserved word onto the stack is emitted, followed by the second last, and so on, ending with an instruction to `push` the first reserved word onto the stack. A `call` instruction with the zeroth reserved word as the operand is then emitted. Note that L2 expects registers `esp`, `ebp`, `ebx`, `esi`, and `edi` to be preserved across `call`s. An `add` instruction that pops N words off the stack is then emitted. Then an instruction is emitted to `mov` the register `eax` into a memory location designated by the surrounding expression.

A function with the reference `-` that returns the value of subtracting its second parameter from its first could be defined as follows:

```
-:
movl 4(%esp), %eax
subl 8(%esp), %eax
ret
```

The following invocation of it, `(invoke putchar (invoke - (literal 0...01100011) (literal 0...01)))`, prints the text "b" to standard output.

With

```
(with continuation0 expression0)
```

Makes a continuation to the containing expression that is to be `jump`ed to with exactly one argument. Then `expression0` is evaluated in an environment where `continuation0` is a reference to the aforementioned continuation. The resulting value of this expression is unspecified if the evaluation of `expression0` completes. If the continuation `continuation0` is `jump`ed to, then this `with` expression evaluates to the resulting value of the single argument within the responsible `jump` expression.

5+1 words must be reserved in the current function's stack-frame plan. Call the reference to the first word of the reservation `continuation0`. This expression is implemented by first emitting instructions to store the program's state at `continuation0`, that is, instructions are emitted to `mov ebp`, the address of the instruction that should be executed after continuing (a label to be emitted later), `edi`, `esi`, and `ebx`, in that order, to the first 5 words at `continuation0`. After this, the instructions for `expression0` are emitted. Then the label for the first instruction of the continuation is emitted. And finally, an instruction is emitted to `mov` the resulting value of the continuation, the 6th word at `continuation0`, into the memory location designated by the surrounding expression.

Examples

Note that the expression `{continuation0 expression0} jump`s to the continuation reference given by `continuation0` with resulting value of evaluating `expression0` as its argument. With the note in mind, the expression `(begin [putchar (with ignore (begin {ignore (literal 0...01001110)} [foo] [foo] [foo]))] [bar])` prints the text "nbar" to standard output.

The following assembly function `allocate` receives the number of bytes it is to allocate as its first argument, allocates that memory, and passes the initial address of this memory as the single argument to the continuation it receives as its second argument.

```
allocate:
/* All sanctioned by L2 ABI: */
movl 8(%esp), %ecx
```

```
movl 16(%ecx), %ebx
movl 12(%ecx), %esi
movl 8(%ecx), %edi
movl 0(%ecx), %ebp
subl 4(%esp), %esp
andl $0xFFFFFFFC, %esp
movl %esp, 20(%ecx)
jmp *4(%ecx)
```

The following usage of it, `(with dest [allocate (literal 0...011) dest])`, evaluates to the address of the allocated memory. If allocate had just decreased `esp` and returned, it would have been invalid because L2 expects functions to preserve `esp`.

Continuation

```
(continuation continuation0 (param1 param2 ... paramN) expression0)
```

Makes a continuation to be `jump`ed to with exactly `N` arguments. When the continuation is `jump`ed to, `expression0` is evaluated in an environment where `continuation0` is a reference to the continuation itself and `param1`, `param2`, up to `paramN` are the resulting values of evaluating the corresponding arguments in the `jump` expression `jump`ing to this function. Undefined behavior occurs if the evaluation of `expression0` completes - i.e. programmer must direct the control flow out of `continuation0` somewhere within `expression0`. The resulting value of this `continuation` expression is a reference to the continuation.

5+N words must be reserved in the current function's stack-frame plan. Call the reference to the first word of the reservation `continuation0`. This expression is implemented by first emitting an instruction to `mov` the reference `continuation0` into the memory location designated by the surrounding expression. Instructions are then emitted to store the program's state at `continuation0`, that is, instructions are emitted to `mov` `ebp`, the address of the instruction that should be executed after continuing (a label to be emitted later), `edi`, `esi`, and `ebx`, in that order, to the first 5 words at `continuation0`. Then an instruction is emitted to `jmp` to the end of all the instructions that are emitted for this `continuation` expression. Then the label for the first instruction of the continuation is emitted. After this the instructions for `expression0` are emitted.

The expression `{(continuation forever (a b) (begin [putchar a] [putchar b] {forever [- a (literal 0...01)] [- b (literal 0...01)]})) (literal 0...01011010) (literal 0...01111010)}` prints the text "ZzYyXxWw"... to standard output.

Jump

```
(jump continuation0 expression1 expression2 ... expressionN)
{continuation0 expression1 expression2 ... expressionN}
```

Both the above expressions are equivalent. Evaluates `continuation0`, `expression1`, `expression2`, up to `expressionN` in an unspecified order and then `jump`s to `continuation0`, a reference to a continuation, providing it with a local copies of `expression1` up to `expressionN` in order. The resulting value of this expression is unspecified.

`N+1` words must be reserved in the current function's stack-frame plan. The expression is implemented by emitting the instructions for any of the subexpressions with the location of the resulting value fixed to the corresponding reserved word. The same is done with the remaining expressions repeatedly until the instructions for all the subexpressions have been emitted. Then an instruction to `mov` the first reserved word to 5 words from the beginning of the continuation is emitted, followed by an instruction to `mov` the second reserved word to an address immediately after that, and so on, ending with an instruction to `mov` the last

reserved word into the last memory address of that area. The program's state, that is, `ebp`, the address of the instruction that should be executed after continuing, `edi`, `esi`, and `ebx`, in that order, are what is stored at the beginning of a continuation. Instructions to `mov` these values from the buffer into the appropriate registers and then set the program counter appropriately are, at last, emitted.

The expression `(begin (with cutter (jump (continuation cuttee () (begin [bar] [bar] (jump cutter (literal 0...0)) [bar] [bar] [bar])))) [foo])` prints the text "barbarfoo" to standard output.

An Optimization

Looking at the examples above where the continuation reference does not escape, `(with reference0 expression0)` behaves a lot like the pseudo-assembly `expression0 reference0:` and `(continuation reference0 (...) expression0)` behaves a lot like `reference0: expression0`. To be more precise, when references to a particular continuation only occur as the `continuation0` subexpression of a `jump` statement, we know that the continuation is constrained to the function in which it is declared, and hence there is no need to store or restore `ebp`, `edi`, `esi`, and `ebx`. Continuations, then, are how efficient iteration is achieved in L2.

Syntactic Sugar

`$a1...aN`

In what follows, it is assumed that `$a1...aN` is not part of a larger string. If `$a1...aN` is simply a `$`, then it remains unchanged. Otherwise at least a character follows the `$`; in this case `$a1...aN` turns into `($ a1...aN)`.

For example, the expression `$$hello$bye` turns into `($ $hello$bye)` which turns into `($ ($ hello$bye))`

`#a1...aN, ,a1...aN, ‘a1...aN`

Analogous transformations to the one for `$a1...aN` happen.

Internal Representation

After substituting out the syntactic sugar defined in the invoke, jump, and syntactic sugar sections, we find that all L2 programs are just fragments where a fragment is either a token or a list of fragments. And furthermore, every token can be seen as a list of its characters so that for example `foo` becomes `(f o o)`. The following functions that manipulate these fragments are not part of the L2 language and hence the compiler does not give references to them special treatment during compilation. However, when they are used in an L2 meta-program, undefined references to these functions are to be resolved by the compiler.

`[lst x y b]`

`y` must be a list and `b` a buffer.

Makes a list where `x` is first and `y` is the rest in the buffer `b`.

Say that `a` is the fragment `foo` and `b` is the list `(bar)`. Then `[lst a b]` is the fragment `(foo bar)`.

[token? x]

x must be a fragment.

Evaluates to the one if x is also a token. Otherwise evaluates to zero.

Say that a is the fragment foo. Then [token? a] evaluates to (literal 0...01).

[@fst x]

x must be a list.

Evaluates to the first of x.

Say that a is the list foo. Then [@fst a] is the character f. This f is not a list but is a character.

[@rst x]

x must be a list.

Evaluates to a list that is the rest of x.

Say that a is the list foo. Then [@rst a] is the fragment oo.

emt

Evaluates to the empty list.

Say that a is the fragment foo. Then [lst a emt] is the fragment (foo).

[emt? x]

x must be a list.

Evaluates to the one if x is the empty list. Otherwise evaluates to zero.

[emt? emt] evaluates to (literal 0...01).

-<character>-

Evaluates to the character <character>.

Say that b is a buffer. Then the expression [lst -f- [lst -o- [lst -o- emt b] b] b] evaluates to the fragment foo.

[char= x y]

x and y must be characters.

Evaluates to one if x is the same character as y, otherwise it evaluates to zero.

Say that x and y are the character d. Then [char= x y] evaluates to (literal 0...01).

`[begin x b]`

`x` must be a list of fragments and `b` a buffer.

Evaluates to an fragment formed by prepending the token `begin` to `x`. The `begin` function could have the following definition: `(function begin (frags b) [lst [lst -b- [lst -e- [lst -g- [lst -i- [lst -n- emt b] b] b] b] b] frags b])`.

`[literal x b]`, `[storage x b]`, `[if x b]`, `[function x b]`, `[invoke x b]`, `[with x b]`, `[continuation x b]`, `[jump x b]`

These functions are analogous to `begin`.

Expressions Continued

Meta

`(function0 expression1 ... expressionN)`

If the above expression is not listed above, then `function0` from the metaprogram is invoked with the (unevaluated) list of fragments `(expression1 expression2 ... expressionN)` as its first argument and a buffer in which the replacement is to be constructed as its second argument. The fragment returned by this function then replaces the entire fragment `(function0 expression1 ... expressionN)`. If the result of this replacement contains a meta-expression, then the above process is repeated. When this process terminates, the appropriate assembly code for the resulting expression is emitted.

Meta-expressions were already demonstrated in the compiler section.

Examples/Reductions

In the extensive list processing that follows in this section, the following functions prove to be convenient abbreviations:

abbreviations.l2

```
(function @frst (l) [@fst [@rst l]])
(function @ffrst (l) [@fst [@frst l]])
(function @frfrst (l) [@fst [@rst [@frst l]]])
(function @rrst (l) [@rst [@rst l]])
(function @rrrst (l) [@rst [@rrst l]])
(function @rfst (l) [@rst [@fst l]])
(function @frfst (l) [@fst [@rfst l]])
(function @frrfst (l) [@fst [@rst [@rfst l]]])
(function @frrst (l) [@fst [@rst [@rst l]]])
(function @frrrst (l) [@fst [@rst [@rst [@rst l]]]])
(function @frrrrst (l) [@fst [@rst [@rst [@rst [@rst l]]]]])
(function @frrrrrst (l) [@fst [@rst [@rst [@rst [@rst [@rst l]]]]]])
(function @ffst (l) [@fst [@fst l]])
(function llst (a b c r) [lst a [lst b c r] r])
(function lllst (a b c d r) [lst a [llst b c d r] r])
```

```
(function llllst (a b c d e r) [lst a [lllst b c d e r] r])
(function lllllst (a b c d e f r) [lst a [llllst b c d e f r] r])
(function lllllllst (a b c d e f g r) [lst a [lllllst b c d e f g r] r])
(function llllllllst (a b c d e f g h r) [lst a [lllllllst b c d e f g h r] r])
```

Commenting

L2 has no built-in mechanism for commenting code written in it. The following comment function takes a list of fragments as its argument and returns an empty begin expression effectively causing its arguments to be ignored. Its implementation and use follows:

comments.l2

```
(function ;; (l r) [lst [llllllst -b- -e- -g- -i- -n- emt r] emt r])
```

test1.l2

```
(;; This is a comment, take no notice.)
```

shell

```
./bin/l2compile "bin/x86_64.o" abbreviations.l2 comments.l2 - test1.l2
```

Numbers

Integer literals prove to be quite tedious in L2 as can be seen from some of the examples in the expressions section. The following function, #, implements decimal arithmetic for x86-64 by reading in a token in base 10 and writing out the equivalent fragment in base 2:

numbers64.l2

```
(;; Turns an 8-byte value into a literal-expression representation of it.)

(function value->literal (binary r)
  [lst [llllllllst -l- -i- -t- -e- -r- -a- -l- emt r]
    [lst (with return {(continuation write (count in out)
        (if count
          {write [- count (literal 0...01)]
            [>> in (literal 0...01)]
            [lst (if [land in (literal 0...01)]
              -1- -0-) out r]}
          {return out}))
        (literal 0...01000000) binary emt})
      emt r]r])

(;; Turns the base-10 fragment input into a literal expression.)

(function # (l r) [value->literal
  (with return {(continuation read (in out)
```

```
    (if [emt? in]
      {return out}
      {read [@rst in] [+ [* out (literal 0...01010)]
        (if [char= [@fst in] -9-] (literal 0...01001)
        (if [char= [@fst in] -8-] (literal 0...01000)
        (if [char= [@fst in] -7-] (literal 0...0111)
        (if [char= [@fst in] -6-] (literal 0...0110)
        (if [char= [@fst in] -5-] (literal 0...0101)
        (if [char= [@fst in] -4-] (literal 0...0100)
        (if [char= [@fst in] -3-] (literal 0...011)
        (if [char= [@fst in] -2-] (literal 0...010)
        (if [char= [@fst in] -1-] (literal 0...01)
          (literal 0...0)))))))))]}))
    [@fst l] (literal 0...0)}) r])
```

test3.l2

```
[putchar (# 65)]
```

or equivalently

```
[putchar #65]
```

shell

```
./bin/l2compile "bin/x86_64.o" abbreviations.l2 comments.l2 numbers64.l2 - test3.l2
```

Backquoting

The `foo` example in the internal representation section shows how tedious writing a function that outputs a token can be. The backquote function reduces this tedium. It takes a fragment and a buffer as its argument and, generally, it returns a fragment that makes that fragment. The exception to this rule is that if a sub-expression of its input fragment is of the form `(, expr0)`, then the fragment `expr0` is inserted verbatim into that position of the output fragment. Backquote can be implemented and used as follows:

backquote.l2

```
(function ` (l r)
  [(function aux (s t r)
    (if [emt? s] [lllst -e- -m- -t- emt r]

    (if (if [emt? s] #0 (if [token? s] #0 (if [emt? [@fst s]]
      #0 (if [char= [@ffst s] -,-] [emt? [@rfst s]] #0))))
          [@frst s]

    [llllllst [lllllllst -i- -n- -v- -o- -k- -e- emt r]
      [lllst -l- -s- -t- emt r]
        (if [token? s]
            [lllst --- [@fst s] --- emt r]
            [aux [@fst s] t r])
```

```
          [aux [@rst s] t r] t emt r])))) [@fst l] [@frst l] r])
```

anotherfunction.l2:

```
(function make-A-function (l r)
  (` (function A (,emt) [putchar #65]) r))
```

or equivalently

```
(function make-A-function (l)
  (`(function A () [putchar #65])r))
```

test4.l2

```
[(make-A-function)]
```

shell

```
./bin/l2compile "bin/x86_64.o" abbreviations.l2 comments.l2 numbers64.l2 backquote.l2 \
  anotherfunction.l2 - test4.l2
```

Variable Binding

Variable binding is enabled by the `continuation` expression. `continuation` is special because, like `function`, it allows identifiers to be bound. Unlike `function`, however, expressions within `continuation` can directly access its parent function's variables. The `let` binding function implements the following transformation:

```
(let (params args) ... expr0)
->
(with let:return
  {(continuation let:aux (params ...)
    {let:return expr0}) vals ...})
```

It is implemented and used as follows:

let.l2

```
(;; Reverses the given list. l is the list to be reversed. r is the buffer into
  which the reversed list will be put. Return value is the reversed list.)

(function meta:reverse (l r)
  (with return
    {(continuation _ (l reversed)
      (if [emt? l]
        {return reversed}
        {_ [@rst l] [lst [@fst l] reversed r]})) l emt}))

(;; Maps the given list using the given function. l is the list to be mapped. ctx
  is always passed as a second argument to the mapper. mapper is the two argument
  function that will be supplied a list item as its first argument and ctx as its
  second argument and will return an argument that will be put into the corresponding
```

```
  position of another list. r is the buffer into which the list being constructed
  will be put. Return value is the mapped list.)

(function meta:map (l ctx mapper r)
  (with return
    {(continuation aux (in out)
      (if [emt? in]
        {return [meta:reverse out r]}
        {aux [@rst in] [lst [mapper [@fst in] ctx] out r]})) l emt}))

(function let (l r)
  (`(with let:return
    (,[llst (` jump r)
      (`(continuation let:aux (,[meta:map [@rst [meta:reverse l r]] (begin) @fst r])
        {let:return (,[@fst [meta:reverse l r]])}) r)
      [meta:map [@rst [meta:reverse l r]] (begin) @frst r] r])) r))
```

test5.l2

```
(let (x #12) (begin
  (function what? () [printf (" x is %i) x])
  [what?]
  [what?]
  [what?]))
```

Note in the above code that `what?` is only able to access `x` because `x` is defined outside of all functions and hence is statically allocated. Also note that L2 permits identifier shadowing, so `let` expressions can be nested without worrying, for instance, about the impact of an inner `templet0` on an outer one.

shell

```
./bin/l2compile "bin/x86_64.o" abbreviations.l2 comments.l2 numbers64.l2 backquote.l2 \
  let.l2 - test5.l2
```

Boolean Expressions

The Boolean literals true and false are achieved using macros that return the same literal fragment regardless of the arguments supplied to them. Short-circut Boolean expressions are achieved through the `if` expression. The `if` expression is special because it has the property that only two out of its three sub-expressions are evaluated when it itself is evaluated. Now, the Boolean expressions implement the following transformations:

```
(false) -> (literal #0)

(true) -> (literal #1)

(or expr1 expr2 ... exprN)
->
(let (or:temp expr1) (if or:temp
  or:temp
  (let (or:temp expr2) (if or:temp
    or:temp
```

```
    ...
      (let (or:temp exprN) (if or:temp
        or:temp
        (false)))))))

(and expr1 expr2 ... exprN)
->
(let (and:temp expr1) (if and:temp
  (let (and:temp expr2) (if and:temp
    ...
      (let (and:temp exprN) (if and:temp
        (true)
        and:temp))
    and:temp))
  and:temp))

(not expr1)
->
(if expr1 (false) (true))
```

These transformations are implemented and used as follows:

boolean.l2

```
(function mk# (r value) [value->literal value r])

(function false (l r) [mk# r #0])

(function true (l r) [mk# r #1])

(function or (l r) (with return
  {(continuation loop (l sexpr)
      (if [emt? l]
        {return sexpr}
        {loop [@rst l] (`(let (or:temp (,[@fst l])) (if or:temp or:temp (, sexpr r)))r)}))
    [meta:reverse l r] (`(false)r)}))

(function and (l r) (with return
  {(continuation loop (l sexpr)
      (if [emt? l]
        {return sexpr}
        {loop [@rst l] (`(let (and:temp (,[@fst l])) (if and:temp (, sexpr r) and:temp))r)}))
    [meta:reverse l r] (`(true)r)}))

(function not (l r) (`(if (,[@fst l]) (false) (true))r))
```

test6.l2

```
(and (false) [/ #1 #0])
```

shell

```
./bin/l2compile "bin/x86_64.o" abbreviations.l2 comments.l2 numbers64.l2 backquote.l2 \
  let.l2 boolean.l2 - test6.l2
```

Switch Expression

Now we will implement a variant of the switch statement that is parameterized by an equality predicate. The `switch` selection function will, for example, do the following transformation:

```
(switch eq0 val0 (vals exprs) ... expr0)
->
(let (tempeq0 eq0) (tempval0 val0)
  (if [tempeq0 tempval0 vals1]
    exprs1
    (if [tempeq0 tempval0 vals2]
      exprs2
      ...
        (if [tempeq0 tempval0 valsN]
          exprsN
          expr0))))
```

It is implemented and used as follows:

switch.l2

```
(function switch (l r)
  (`(let (switch:= (,[@fst l])) (switch:val (,[@frst l]))
    (,(with return
      {(continuation aux (remaining else-clause)
        (if [emt? remaining]
          {return else-clause}
          {aux [@rst remaining]
            (`(if (,[lst (` or r) [meta:map [@rst [meta:reverse [@fst remaining] r]] r
                (function _ (e r)
                  [llllst (` invoke r) (` switch:= r) (` switch:val r) e emt r]) r] r])
              (,[@fst [meta:reverse [@fst remaining] r]]) ,else-clause) r)}))
        [@rst [meta:reverse [@rrst l] r]] [@fst [meta:reverse l r]]}))r))
```

test7.l2

```
(switch = #10
  (#20 [printf (" d is 20!)])
  (#10 [printf (" d is 10!)])
  (#30 [printf (" d is 30!)])
  [printf (" s is something else.)])
```

shell

```
./bin/l2compile "bin/x86_64.o" abbreviations.l2 comments.l2 numbers64.l2 backquote.l2 \
  let.l2 boolean.l2 switch.l2 - test7.l2
```

Characters

With `#` implemented, a somewhat more readable implementation of characters is possible. The `char` function takes a singleton list containing a token of one character and returns its ascii encoding using the `#` expression. Its implementation and use follows:

characters.l2

```
(function char (l r) (switch char= [@ffst l]
  (-!- (` #33 r)) (-"- (` #34 r)) (-#- (` #35 r)) (-$- (` #36 r)) (-%- (` #37 r))
  (-&- (` #38 r)) (-'- (` #39 r)) (-*- (` #42 r)) (-+- (` #43 r)) (-,- (` #44 r))
  (--- (` #45 r)) (-.- (` #46 r)) (-/- (` #47 r)) (-0- (` #48 r)) (-1- (` #49 r))
  (-2- (` #50 r)) (-3- (` #51 r)) (-4- (` #52 r)) (-5- (` #53 r)) (-6- (` #54 r))
  (-7- (` #55 r)) (-8- (` #56 r)) (-9- (` #57 r)) (-:- (` #58 r)) (-;- (` #59 r))
  (-<- (` #60 r)) (-=- (` #61 r)) (->- (` #62 r)) (-?- (` #63 r)) (-@- (` #64 r))
  (-A- (` #65 r)) (-B- (` #66 r)) (-C- (` #67 r)) (-D- (` #68 r)) (-E- (` #69 r))
  (-F- (` #70 r)) (-G- (` #71 r)) (-H- (` #72 r)) (-I- (` #73 r)) (-J- (` #74 r))
  (-K- (` #75 r)) (-L- (` #76 r)) (-M- (` #77 r)) (-N- (` #78 r)) (-O- (` #79 r))
  (-P- (` #80 r)) (-Q- (` #81 r)) (-R- (` #82 r)) (-S- (` #83 r)) (-T- (` #84 r))
  (-U- (` #85 r)) (-V- (` #86 r)) (-W- (` #87 r)) (-X- (` #88 r)) (-Y- (` #89 r))
  (-Z- (` #90 r)) (-\- (` #92 r)) (-^- (` #94 r)) (-_- (` #95 r)) (-`- (` #96 r))
  (-a- (` #97 r)) (-b- (` #98 r)) (-c- (` #99 r)) (-d- (` #100 r)) (-e- (` #101 r))
  (-f- (` #102 r)) (-g- (` #103 r)) (-h- (` #104 r)) (-i- (` #105 r)) (-j- (` #106 r))
  (-k- (` #107 r)) (-l- (` #108 r)) (-m- (` #109 r)) (-n- (` #110 r)) (-o- (` #111 r))
  (-p- (` #112 r)) (-q- (` #113 r)) (-r- (` #114 r)) (-s- (` #115 r)) (-t- (` #116 r))
  (-u- (` #117 r)) (-v- (` #118 r)) (-w- (` #119 r)) (-x- (` #120 r)) (-y- (` #121 r))
  (-z- (` #122 r)) (-|- (` #124 r)) (-~- (` #126 r)) (` #0 r)))
```

test8.l2

```
[putchar (char A)]
```

shell

```
./bin/l2compile "bin/x86_64.o" abbreviations.l2 comments.l2 numbers64.l2 backquote.l2 \
  let.l2 boolean.l2 switch.l2 characters.l2 - test8.l2
```

Strings

The above exposition has purposefully avoided making strings because it is tedious to individually store each character literal in memory. The quote function takes a list of tokens and returns the sequence of operations required to write its ascii encoding into memory. (An extension to this rule occurs when instead of a token, a fragment that is a list of fragments is encountered. In this case the value of the fragment is taken as the character to be inserted.) These "operations" are essentially reserving enough storage for the bytes of the input, putting the characters into that memory, and returning the address of that memory. Because the stack-frame of a function is destroyed upon its return, strings implemented in this way should not be returned. Quote is implemented below:

strings.l2

```
(function " (l r) (with return
  {(continuation add-word (str index instrs)
    (if [emt? str]
      {return (`(with dquote:return
        (,[llst (` begin r) [llst (` storage r) (` dquote:str r)
            (with return {(continuation _ (phs num)
              (if num
                {_ [lst (` #0 r) phs r] [- num #1]}
                {return phs})) emt [+[/ index (unit)]#1]}) r]
          [meta:reverse [lst (`{dquote:return dquote:str}r) instrs r]r]r]))r)}

    (if (and [emt? [@fst str]] [emt? [@rst str]])
      {add-word [@rst str] [+ index #1]
        [lst (`[setb [+ dquote:str (,[value->literal index r])] #0]r) instrs r]}

    (if (and [emt? [@fst str]] [token? [@frst str]])
      {add-word [@rst str] [+ index #1]
        [lst (`[setb [+ dquote:str (,[value->literal index r])] #32]r) instrs r]}

    (if [emt? [@fst str]] {add-word [@rst str] index instrs}

    (if [token? [@fst str]]
      {add-word [lst [@rfst str] [@rst str] r] [+ index #1]
        [lst (`[setb [+ dquote:str (,[value->literal index r])]
          (,[char [lst [lst [@ffst str] emt r] emt r]r emt])]r) instrs r]}

      {add-word [@rst str] [+ index #1]
        [lst (`[setb [+ dquote:str (,[value->literal index r])] (,[@fst str])]r)
          instrs r]})))))) l #0 emt}))
```

test9.l2

```
[printf (" This is how the quote macro is used. (# 10) Now we are on a new line because 10
  is a line feed.)]
```

shell

```
./bin/l2compile "bin/x86_64.o" abbreviations.l2 comments.l2 numbers64.l2 backquote.l2 \
  let.l2 boolean.l2 switch.l2 characters.l2 strings.l2 - test9.l2
```

Closures

A restricted form of closures can be implemented in L2. The key to their implementation is to `jump` out of the function that is supposed to provide the lexical environment. By doing this instead of merely returning from the environment function, the stack-pointer and thus the stack-frame of the environment are preserved. The following example implements a function that receives a single argument and "returns" (more accurately: jumps out) a continuation that adds this value to its own argument. But first, the following transformations are needed:

```
(lambda (args ...) expr0)
->
(continuation lambda0 (cont0 args ...)
  {cont0 expr0})

(; func0 args ...)
->
(with return [func0 return args ...])

(: cont0 args ...)
->
(with return {cont0 return args ...})
```

These are implemented and used as follows:

closures.l2

```
(function lambda (l r)
  (`(continuation lambda0 (,[lst (` cont0 r) [@fst l] r])
    {cont0 (,[@frst l])})r))

(function ; (l r)
  (`(with semicolon:return
    (,[lllst (` invoke r) [@fst l] (` semicolon:return r) [@rst l] r]))r))

(function : (l r)
  (`(with colon:return (,[lllst (` jump r) [@fst l] (` colon:return r) [@rst l] r]))r))
```

test10.l2

```
(function adder (cont x)
  {cont (lambda (y) [+ x y])})

(let (add5 (; adder #5)) (add7 (; adder #7))
  (begin
    [printf (" %i,) (: add5 #2)]
    [printf (" %i,) (: add7 #3)]
    [printf (" %i,) (: add5 #1)]))
```

shell

```
./bin/l2compile "bin/x86_64.o" abbreviations.l2 comments.l2 numbers64.l2 backquote.l2 \
  let.l2 boolean.l2 switch.l2 characters.l2 strings.l2 closures.l2 - test10.l2
```

Assume

There are far fewer subtle ways to trigger undefined behaviors in L2 than in other unsafe languages because L2 does not have dereferencing, arithmetic operators, types, or other such functionality built in; the programmer has to implement this functionality themselves in assembly routines callable from L2. This shift in responsibility means that any L2 compiler is freed up to treat invocations of undefined behaviors in L2 code as intentional.

The following usage of undefined behavior within the function `assume` is inspired by Regehr. The function `assume`, which compiles `y` assuming that the condition `x` holds, implements the following transformation.

```
(assume x y)
->
(with return
  {(continuation tempas0 ()
    (if x {return y} (begin)))})
```

This is implemented as follows:

assume.l2

```
(function assume (l r)
  (`(with assume:return
    {(continuation assume:tempas0 ()
      (if (,[@fst l]) {assume:return (,[@frst l])} (begin)))})r))
```

test11.l2

```
(function foo (x y)
  (assume [not [= x y]] (begin
    [setb x (char A)]
    [setb y (char B)]
    [printf (" %c) [getb x]])))

[foo (" C) (" D)]
```

In the function `foo`, if `x` were equal to `y`, then the else branch of the `assume`'s `if` expression would be taken. Since this branch does nothing, `continuation`'s body expression would finish evaluating. But this is the undefined behavior stated in the first paragraph of the description of the `continuation` expression. Therefore an L2 compiler does not have to worry about what happens in the case that `x` equals `y`. In light of this and the fact that the `if` condition is pure, the whole `assume` expression can be replaced with the first branch of `assume`'s `if` expression. And more importantly, the the first branch of `assume`'s `if` expression can be optimized assuming that `x` is not equal to `y`. Therefore, a hypothetical optimizing compiler would also replace the last `[getb x]`, a load from memory, with `(char A)`, a constant.

shell

```
./bin/l2compile "bin/x86_64.o" abbreviations.l2 comments.l2 numbers64.l2 backquote.l2 \
  let.l2 boolean.l2 switch.l2 characters.l2 strings.l2 assume.l2 - test11.l2
```

Note that the `assume` expression can also be used to achieve C's `restrict` keyword simply by making its condition the conjunction of inequalities on the memory locations of the extremeties of the "arrays" in question.

Fields

L2 has no built-in mechanism for record and union types. The most naive way to do record types in L2 would be to create a getter function, setter function, and offset calculation function for every field where these functions simply access and/or mutate the desired memory locations. However this solution is untenable because of the amount of boilerplate that one would have to write. A better solution is to aggregate the

offset, size, getter, and setter of each field into a higher-order macro that supplies this information into any macro that is passed to it. This way, generic getter, setter, address-of, offset-of, and sizeof functions can be defined once and used on any field. More concretely, the following transformations are what we want:

```
(offset-of expr0)
->
(expr0 offset-aux)

(offset-aux expr0 ...)
->
expr0

(size-of expr0)
->
(expr0 size-of-aux)

(size-of-aux expr0 expr1 ...)
->
expr1

(getter-of expr0)
->
(expr0 getter-of-aux)

(getter-of-aux expr0 expr1 expr2 ...)
->
expr2

(setter-of expr0)
->
(expr0 setter-of-aux)

(setter-of-aux expr0 expr1 expr2 expr3 ...)
->
expr3

(& expr0 expr1)
->
(expr0 &-aux expr1)

(&-aux expr0 expr1 expr2 expr3 expr4 ...)
->
[+ expr4 expr0]

(@ expr0 expr1)
->
(expr0 @-aux expr1)

(@-aux expr0 expr1 expr2 expr3 expr4 ...)
->
[expr2 [+ expr4 expr0]]
```

```
(setf expr0 expr1 expr2)
->
(expr0 setf-aux expr1 expr2)

(setf-aux expr0 expr1 expr2 expr3 expr4 expr5)
->
[expr3 [+ expr4 expr0] expr5]
```

Why? Because if we define the macro `car` by the transformation `(car expr0 exprs ...) -> (expr0 #0 #8 get8b set8b exprs ...)` and `cdr` by the transformation `(cdr expr0 exprs ...) -> (expr0 #8 #8 get8b set8b exprs ...)`, then we get the following outcomes:

```
(offset-of car) -> (car offset-of-aux) -> (offset-of-aux #0 #8 get8b set8b) -> #0
(size-of car) -> (car size-of-aux) -> (size-of-aux #0 #8 get8b set8b) -> #8
(getter-of car) -> (car getter-of-aux) -> (getter-of-aux #0 #8 get8b set8b) -> get8b
(setter-of car) -> (car setter-of-aux) -> (setter-of-aux #0 #8 get8b set8b) -> set8b
(& car expr) -> (car &-aux expr) -> (&-aux #0 #8 get8b set8b expr) -> [+ expr #0]
(@ car expr) -> (car @-aux expr) -> (@-aux #0 #8 get8b set8b expr) -> [get8b [+ expr #0]]
(setf car expr val) -> (car setf-aux expr val) -> (setf-aux #0 #8 get8b set8b expr val) ->
  [set8b [+ expr #0] val]

(offset-of cdr) -> ... -> #8
(size-of cdr) -> ... -> #8
(getter-of cdr) -> ... -> get8b
(setter-of cdr) -> ... -> set8b
(& cdr expr) -> ... -> [+ expr #8]
(@ cdr expr) -> ... -> [get8b [+ expr #8]]
(setf cdr expr val) -> ... -> [set8b [+ expr #8] val]
```

To recapitulate, we localized and separated out the definition of a field from the various operations that can be done on it. Since dozens of fields can potentially be used in a program, it makes sense to define a helper function, `mk-field`, that creates them. What follows is the implementation of this helper function and the aforementioned transformations:

fields.l2

```
(function offset-of (l r) (`((,[@fst l]) offset-of-aux)r))

(function offset-of-aux (l r) [@fst l])

(function size-of (l r) (`((,[@fst l]) size-of-aux)r))

(function size-of-aux (l r) [@frst l])

(function getter-of (l r) (`((,[@fst l]) getter-of-aux)r))

(function getter-of-aux (l r) [@frrst l])

(function setter-of (l r) (`((,[@fst l]) setter-of-aux)r))

(function setter-of-aux (l r) [@frrrst l])

(function & (l r) (`((,[@fst l]) &-aux (,[@frst l]))r))
```

```
(function &-aux (l r) (`[+ (,[@frrrrst l]) (,[@fst l])]r))

(function @ (l r) (`((,[@fst l]) @-aux (,[@frst l]))r))

(function @-aux (l r) (`[(,[@frrst l]) [+ (,[@frrrrst l]) (,[@fst l])]]r))

(function setf (l r) (`((,[@fst l]) setf-aux (,[@frst l]) (,[@frrst l]))r))

(function setf-aux (l r)
  (`[(,[@frrrst l]) [+ (,[@frrrrst l]) (,[@fst l])] (,[@frrrrrst l])]r))

(function mk-field (l r offset size)
  [llllllst [@fst l] [value->literal offset r] [value->literal size r]
    (switch = size
      (#1 (` get1b r)) (#2 (` get2b r)) (#4 (` get4b r)) (#8 (` get8b r)) (`(begin)r))
    (switch = size
      (#1 (` set1b r)) (#2 (` set2b r)) (#4 (` set4b r)) (#8 (` set8b r)) (`(begin)r))
    [@rst l] r])
```

somefields.l2

```
(function cons-cell (l r) [mk# r #16])

(function car (l r) [mk-field l r #0 #8])

(function cdr (l r) [mk-field l r #8 #8])
```

test12.l2

```
(storage mycons (begin) (begin))
(setf car mycons (char A))
(setf cdr mycons (char a))
[putchar (@ car mycons)]
[putchar (@ cdr mycons)]
```

shell

```
./bin/l2compile "bin/x86_64.o" abbreviations.l2 comments.l2 numbers64.l2 backquote.l2 \
  let.l2 boolean.l2 switch.l2 characters.l2 strings.l2 assume.l2 fields.l2 somefields.l2 - \
  test12.l2
```

Note that there is no struct definition in the code, there are only definitions of the fields we need to work with. The negative consequence of this is that we lose C's type safety and portability. The positive consequences are that we gain control over the struct packing, we are now able to use the same field definitions across several conceptually different structs, and that we can overlap our fields in completely arbitrary ways.