L2

Murisi Tarusenga

Thursday 26th October, 2017 15:40

1 Introduction

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 preprocessor directives, control structures, statements, literals, and functions requiring compiler assistance (setjmp, longjmp, alloca, 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 reckon the result is still pretty interesting.

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 brief description of L2's internal representation and the 9 functions (loosely speaking) that manipulate it. After that comes a description of how a non-primitive L2 expression is compiled. The above descriptions take about 8 pages and are essentially a complete description of L2.

Afterwards, 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. And, finally, this README ends with a description of my L2 system's compilation library, a binary interface for compiling L2 code.

Contents

1 Introduction 1

2 Getting Started 2

2.1 Building L2 . . . 2

2.2 The Evaluator . . . 2

2.2.1 Example . . . 3

3 Primitive Expressions 4

3.1 Begin . . . 4

3.2 Binary . . . 4

3.3 Reference . . . 4

3.4 If . . . 4

3.5 Function . . . 5

3.6 Invoke . . . 5

3.7 With . . . 6

3.7.1 Examples . . . 6
3.8 Continuation . . . 6
3.9 Jump . . . 7
3.9.1 An Optimization . . . 7

4 Internal Representation **7**
4.1 [lst x y] . . . 8
4.2 [lst ? x] . . . 8
4.3 [fst x] . . . 8
4.4 [rst x] . . . 8
4.5 [sexpr x] . . . 8
4.6 [nil] . . . 8
4.7 [nil ? x] . . . 8
4.8 [−<character>−] . . . 9
4.9 [<character>? x] . . . 9

5 Expression **9**

6 Examples/Reductions **9**
6.1 Commenting . . . 9
6.2 Numbers . . . 10
6.3 Backquoting . . . 11
6.4 Characters . . . 11
6.5 Strings . . . 13
6.6 Conditional Compilation . . . 14
6.7 Variable Binding . . . 14
6.8 Switch Expression . . . 15
6.9 Closures . . . 15
6.10 Assume . . . 16

7 Compilation Library **17**
7.1 [library x] . . . 17
7.2 [sequence x y] . . . 18
7.3 [executable x] . . . 18
7.4 [copy x y] . . . 18
7.5 [nil −library] . . . 18
7.6 [skip x y] . . . 18
7.7 [concatenate x y] . . . 18
7.8 [nil −source] . . . 18
7.9 [load x] . . . 19
7.10 [unload x] . . . 19
7.11 [dynamic x] . . . 19

2 Getting Started

2.1 Building L2

```
./ buildl2
```

My L2 system needs a Linux distribution running on the i386 (or AMD64 with libc6-dev-i386 installed) architecture with the GNU C compiler installed to run successfully. To build the system, simply run the buildl2 script at the root of the repository. The build should be fast - there are only about 2300 lines of C code to compile. This will create a directory called bin containing the files l2evaluate, l2compile.a, and i386 .a. l2evaluate is an evaluator of L2 code: it reads in L2 code, compiles it, then executes it. l2compile.a is the library that the evaluator uses to compile L2 code. i386.a is a library of instruction wrappers to provide i386 functionality (ADD, SUB, MOV, ...) not exposed by the L2 language.

2.2 The Evaluator

```
./bin/l2evaluate libraries.a ... (− inputs.l2 ...)
    ... − inputs.l2 ...
```

The initial environment, the one that is there before anything is evaluated, comprises 17 functions: lst , lst ?, fst , rst, sexpr, nil , nil ?, −<character >−, <character>?, begin, b, if , function, invoke, with, continuation, and jump. The former 9 are defined later. Each one of the latter 8 functions does nothing else but return an s-expression formed by prepending its function name to the list of s-expressions supplied to them. For example, the b function could have the following definition: (function b (sexprs) [lst [lst [−b−] [nil]] [' sexprs]]).

In this initial environment the L2 evaluator begins by essentially generating machine code for a function whose's behavior is the "concatenation" of the static libraries libraries .a ... , where static library here means the ordered "concatenation" of the object files (or more precisely, their code segments) within it. This generated code is then loaded into memory, and invoked. Afterwords, the memory reserved for the static libraries is maintained, and the libraries' exported functions are also made available for use throughout the remaining evaluations - we will say that the remaining evaluations happen in this augmented environment.

For each hyphen argument, the compiler reads the following inputs.l2 ... until either the next hyphen argument is found or the command line arguments are finished. Each of the files read should be of the form expression1 expression2 ... expressionN. As the evaluator goes through the files in order, it compiles each expression in the current environment and emits the corresponding machine code in the same order as the expressions of the concatenated file. The amalgamated object code is then loaded into memory and executed. The evalutaion of the next set of L2 files happens in this further augmented environment.

If the entire list of arguments ends with a hyphen, then a minus-prompt is displayed. You are expected to enter valid L2 source code. Each time you press Enter, a plus-prompt is printed to elicit you to add to the source code so far read. Pressing Ctrl-D on an empty line tells the evaluator to compile in the environment all the code entered since the last minus prompt, execute it, and augment the environment. The evaluator then returns to minus-prompt state. If Ctrl-C, then Enter is typed in a minus-prompt, the evaluator exits. If this is done instead in a plus-prompt, heretofore uncompiled code is discarded, and the evaluator returns to minus-prompt state.

2.2.1 Example

Listing 1: file1.l2

```
(function foo (sexprs)
  (with return (begin
    [putchar [+ (b 00000000000000000000000001100001)
        (b 00000000000000000000000000000001)]]
    {return [lst [lst [−b−] [lst [−e−] [lst [−g−] [
        lst [−i−] [lst [−n−] [nil]]]]]] [nil]]})))

[putchar (b 00000000000000000000000001100001)]
```

Listing 2: file2.l2

```
(function bar ()
  [putchar
    (b 00000000000000000000000001100011)])
(foo this text does not matter)
[putchar
  (b 00000000000000000000000001100100)]
```

Running ./bin/l2evaluate bin/i386.a − file1 .l2 − file2 .l2 should cause the text "abd" to be printed to standard output. The "a" comes from the last expression of file1.l2. It was printed after the compilation of file1.l2, when it was being loaded into the compiler. Why? Because L2 libraries are executed from top to bottom when they are dynamically loaded (and also when they are statically linked). The "b" comes from within the function in file1.l2. It was executed when the expression (foo this text does not matter) in file2.l2 was being compiled. Why? Because the foo causes the compiler to invoke a function called foo in the environment. The s-expression (this text does not matter) is the argument to the function foo, but the function foo ignores it and returns the s-expression (begin). Hence (begin) replaces (foo this text does not matter) in file2 .l2. Now file2 .l2 is entirely made up of primitive expressions which are compiled in the way specified below. Finally the text "d" is printed. Why? Because file2.l2's last expression is the only one that is side-effectual, does not get replaced during compilation, and (therefore) gets loaded into memory.

Running ./bin/l2evaluate bin/i386.a − file1 .l2 − should cause the evaluator to print "a" to standard output for the same reason as above. Now you should be in a minus-prompt. Pasting in the contents of file2 .l2, then pressing Enter followed by Ctrl-D should now cause the evaluator to print the text "bd" for the same reasons as above.

3 Primitive Expressions

3.1 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.

3.2 Binary

```
(b b31b30...b0)
```

The resulting value is the 32 bit number specified in binary inside the brackets. Specifying less than or more than 32 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 (b 00000000000000000000000001100001)] prints the text "a" to standard output.

3.3 Reference

```
reference0
```

The resulting value is the address in memory to which this reference refers.

This expression is implemented by the emission of an instruction to lea of some data into a memory location designated by the surrounding expression.

Say the expression [' x] evaluates to the value at the reference x and the expression [set x y] puts the value y into the reference x. Then (begin [set x (b 00000000000000000000000001100001)] [putchar [' x]]) prints the text "a" to standard output.

3.4 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 (b 00000000000000000000000000000000)(b 00000000000000000000000001100001)(b 00000000000000000000000001100010)

)] prints the text ”b” to standard output.

3.5 Function

```
(function function0 (reference1 reference2 ...
    referenceN) 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 reference1, reference2, up to referenceN are references to 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]]) (b 00000000000000000000000000000001)(b 00000000000000000000000001100011)]] prints the text ”b” to standard output.

3.6 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 calls. 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 − (b 00000000000000000000000001100011)(b

00000000000000000000000000000001))), prints the text "b" to standard output.

3.7 With

```
(with continuation0 expression0)
```

Makes a continuation to the containing expression that is to be jumped 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 jumped 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.

3.7.1 Examples

Note that the expression {continuation0 expression0} jumps 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 (b 00000000000000000000000001001110)} [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 (b 00000000000000000000000000000011)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.

3.8 Continuation

```
(continuation continuation0 (reference1 reference2
    ... referenceN) expression0)
```

Makes a continuation to be jumped to with exactly N arguments. When the continuation is jumped to, expression0 is evaluated in an environment where continuation0 is a reference to the continuation itself and reference1, reference2, up to referenceN are references to the resulting values of evaluating the corresponding arguments in the jump expression jumping 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] (b 00000000000000000000000000000001)] [− [' b] (b 00000000000000000000000000000001)]}))(b 00000000000000000000000001011010)(b 00000000000000000000000001111010)} prints the text "ZzYyXxWw"... to standard output.

3.9 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 jumps 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 (b 00000000000000000000000000000000))[bar] [bar] [bar])))) [foo]) prints the text "barbarfoo" to standard output.

3.9.1 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.

4 Internal Representation

After substituting out the syntactic sugar used for the invoke and jump expressions. We find that all L2 programs are just compositions of the <pre−s−expression>s: <symbol> and (<pre−s−expression> <pre−s−expression> ... <pre−s−expression>). If we now replace every symbol with a list of its characters so that for example foo becomes (f o o), we now find that all L2 programs are now just compositions of the <s−expression>s <character> and (<s−expression> <s−expression > ... <s−expression>). The following functions that manipulate these s-expressions are not part of the L2 language and hence the compiler does not give references to them special treatment during compilation. However, when compiled code is loaded into an L2 compiler, undefined references to these functions are to be dynamically resolved.

4.1 [lst x y]

x must be a s-expression and y a list.

Makes a list where x is first and y is the rest.

Say the s-expression foo is stored at a and the list (bar) is stored at b. Then [lst [' a] [' b]] is the s-expression (foo bar).

4.2 [lst ? x]

x must be a s-expression.

Evaluates to the complement of zero if x is also list. Otherwise evaluates to zero.

Say the s-expression foo is stored at a. Then [lst ? [' a]] evaluates to (b 11111111111111111111111111111111).

4.3 [fst x]

x must be a list.

Evaluates to a s-expression that is the first of x.

Say the list foo is stored at a. Then [fst [' a]] is the s-expression a. This a is not a list but is a character.

4.4 [rst x]

x must be a list.

Evaluates to a list that is the rest of x.

Say the list foo is stored at a. Then [rst [' a]] is the s-expression oo.

4.5 [sexpr x]

x must be a list.

Evaluates to an s-expression wrapper of x.

Say the s-expression foo is stored at a and (bar) is stored at b. Then [lst [sexpr [rst [' a]]] [' b]] is the s-expression (oo bar). Note that without the sexpr invocation, the preconditions of lst would be violated.

4.6 [nil]

Evaluates to the empty list.

Say the s-expression foo is stored at a. Then [lst [' a] [nil]] is the s-expression (foo).

4.7 [nil ? x]

x must be a list.

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

Say the s-expression ((foo bar bar bar)) is stored at x. Then [nil ? [rst [' x]]] evaluates to (b 11111111111111111111111111111111).

4.8 [−<character>−]

Evaluates to the character <character>.

The expression [lst [−f−] [lst [−o−] [lst [−o−] [nil]]]] evaluates to the s-expression foo.

4.9 [<character>? x]

x must be a s-expression.

Evaluates to the complement of zero if x is the character ¡character¿. Otherwise evaluates to zero.

Say the s-expression (foo (bar bar) foo foo) is stored at x. Then [m? [' x]] evaluates to (b 00000000000000000000000000000000).

5 Expression

```
(function0 expression1 ... expressionN)
```

If the above expression is not a primitive expression, then function0 is evaluated in the environment. The resulting value of this evaluation is then invoked with the (unevaluated) list of s-expressions (expression1 expression2 ... expressionN) as its only argument. The list of s-expressions returned by this function then replaces the entire list of s-expressions (function0 expression1 ... expressionN). If the result of this replacement is still a non-primitive expression, then the above process is repeated. When this process terminates, the appropriate assembly code for the resulting primitive expression is emitted.

The expression ((function comment (sexprs)[fst [' sexprs]]) [foo] This comment is ignored. No, seriously.) is replaced by [foo], which in turn compiles into assembly similar to what is generated for other invoke expressions.

6 Examples/Reductions

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

Listing 3: abbreviations.l2

```
(function frst (l) [fst [rst [' l]]])
(function frfst (l) [fst [rst [fst [' l]]]])
(function frrst (l) [fst [rst [rst [' l]]]])
(function frrrst (l) [fst [rst [rst [rst [' l]]]]])
(function rfst (l) [rst [fst [' l]]])
(function ffst (l) [fst [fst [' l]]])
(function llst (a b c) [lst [' a] [lst [' b] [' c]]])
(function lllst (a b c d) [lst [' a] [lst [' b] [lst
    [' c] [' d]]]])
(function llllst (a b c d e) [lst [' a] [lst [' b] [
    lst [' c] [lst [' d] [' e]]]]])
(function llllllst (a b c d e f g) [lst [' a] [lst ['
    b] [lst [' c] [lst [' d] [lst [' e] [lst [' f] ['
    g]]]]]]])
```

6.1 Commenting

L2 has no built-in mechanism for commenting code written in it. The following comment function that follows takes a list of s-expressions as its argument and returns the last s-expression in that list (which itself is guaranteed to be a list of s-expressions) effectively causing the other s-expressions to be ignored. Its implementation and use follows:

Listing 4: comments.l2

```
(function ** (l)
  (with return
    {(continuation find (first last)
      (if [nil? [' last]]
        {return [' first]}
        {find [fst [' last]] [rst [' last]]})) [fst
           [' l]] [rst [' l]]}))
```

Listing 5: test1.l2

```
(** This is a comment, and the next thing is what is
    actually compiled: (begin))
```

Listing 6: shell

```
./bin/l2evaluate bin/i386.a - abbreviations.l2
    comments.l2 - test1.l2
```

6.2 Numbers

Integer literals prove to be quite tedious in L2 as can be seen from some of the examples in the primitive expressions section. The following function, d, implements decimal arithmetic by reading in an s-expression in base 10 and writing out the equivalent s-expression in base 2:

Listing 7: numbers.l2

```
(** Turns a 4-byte integer into base-2 s-expression
    representation of it.
(function binary->base2sexpr (binary)
  [lst [lst [-b-] [nil]] [lst (with return
    {(continuation write (count in out)
      (if [' count]
        {write [- [' count] (b
            00000000000000000000000000000001)]
          [>> [' in] (b
              00000000000000000000000000000001)]
          [lst (if [and [' in] (b
              00000000000000000000000000000001)] [-1-]
               [-0-]) [' out]]}
        {return [' out]})) (b
            00000000000000000000000000100000) ['
            binary] [nil]}) [nil]]]))

(function d (l)
  [binary->base2sexpr
    (** Turns the base-10 s-expression input into a
        4-byte integer.
      (with return {(continuation read (in out)
        (if [nil? [' in]]
          {return [' out]}
          {read [rst [' in]] [+ [* [' out] (b
              00000000000000000000000000001010)]
            (if [9? [fst [' in]]] (b
                00000000000000000000000000001001)
            (if [8? [fst [' in]]] (b
                00000000000000000000000000001000)
            (if [7? [fst [' in]]] (b
                00000000000000000000000000000111)
            (if [6? [fst [' in]]] (b
                00000000000000000000000000000110)
            (if [5? [fst [' in]]] (b
                00000000000000000000000000000101)
            (if [4? [fst [' in]]] (b
                00000000000000000000000000000100)
            (if [3? [fst [' in]]] (b
                00000000000000000000000000000011)
            (if [2? [fst [' in]]] (b
                00000000000000000000000000000010)
            (if [1? [fst [' in]]] (b
                00000000000000000000000000000001)
              (b 00000000000000000000000000000000))))
                 )))))]})) [fst [' l]] (b
                 00000000000000000000000000000000)}))
                 ])
```

Listing 8: test2.l2

```
[putchar (d 65)]
```

Listing 9: shell

```
./bin/l2evaluate bin/i386.a - abbreviations.l2
    comments.l2 - numbers.l2 - test2.l2
```

6.3 Backquoting

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

Listing 10: backquote.l2

```
(function ` (l)
  [(function aux (s)
    (if [nil? [' s]]
      [lst [sexpr [llllllst [-i-][-n-][-v-][-o-][-k
          -][-e-][nil]]]
        [lst [sexpr [lllst [-n-][-i-][-l-][nil]]] [
            nil]]]

    (if (if [lst? [' s]] (if [not [nil? [' s]]] (if [
        lst? [fst [' s]]] (if [not [nil? [fst [' s]]]]
      (if [,? [ffst [' s]]] [nil? [rfst [' s]]] (d 0)
        ) (d 0)) (d 0)) (d 0)) (d 0))
          [frst [' s]]

    [lst [sexpr [llllllst [-i-][-n-][-v-][-o-][-k-][-
        e-][nil]]]
      [lst [sexpr [lllst [-l-][-s-][-t-][nil]]]
        [lst (if [lst? [fst [' s]]]
          [sexpr [aux [fst [' s]]]]
          [sexpr [lst
            [sexpr [llllllst [-i-][-n-][-v-][-o-][-k
                -][-e-][nil]]]
              [lst [sexpr [lst [---] [lst [fst [' s]]
                  [lst [---] [nil]]]]] [nil]]]]])
            [lst [sexpr [aux [rst [' s]]]] [nil]]]]]])
              )) [fst [' l]]])
```

Listing 11: anotherfunction.l2

```
(function make-A-function (l)
  (` (function A (,[nil]) [putchar (d 65)])))
```

Listing 12: or equivalently

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

Listing 13: test3.l2

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

Listing 14: shell

```
./bin/l2evaluate bin/i386.a - abbreviations.l2
    comments.l2 - numbers.l2 - backquote.l2 -
    anotherfunction.l2 - test3.l2
```

6.4 Characters

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

Listing 15: characters.l2

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

Listing 16: test4.l2

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

Listing 17: shell

```
./bin/l2evaluate bin/i386.a - abbreviations.l2
   comments.l2 - numbers.l2 - backquote.l2 -
   characters.l2 - test4.l2
```

6.5 Strings

The above exposition has purposefully avoided making strings because it is tedious to do using only binary and reference arithmetic. The quote function takes a list of lists of character s-expressions and returns the sequence of operations required to write its ascii encoding into memory. These "operations" are essentially decreasing the stack-pointer, 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 along with its helper function called reverse that reverses lists:

Listing 18: reverse.l2

```
(function reverse (l)
  (with return
    {(continuation _ (l reversed)
      (if [nil? [' l]]
        {return [' reversed]}
        {_ [rst [' l]] [lst [fst [' l]] [' reversed
           ]]})) [' l] [nil]}))
```

Listing 19: strings.l2

```
(function " (l) (with return
  {(continuation add-word (str index instrs)
    (if [nil? [' str]]
      {return (`(with return
        [allocate (,[binary->base2sexpr [' index]])
          (continuation _ (str) (,[lst (` begin) [
             reverse [lst (`{return [' str]}) ['
             instrs]]]]))]))}

      {(continuation add-char (word index instrs)
          (if [nil? [' word]]
            {add-word [rst [' str]] [+ [' index] (d
               1)]
              [lst (`[set-char [+ [' str] (,[binary->
                 base2sexpr [' index]])]
                (,(if [nil? [rst [' str]]] (`(d 0))
                   (`(d 32))))]) [' instrs]]}
            {add-char [rst [' word]] [+ [' index] (d
               1)]
```

```
                [lst ('[set-char [+ [' str] (,[binary->
                    base2sexpr [' index]])] (char (,[lst
                      [fst [' word]] [nil]]))])
                  [' instrs]]}))
        [fst [' str]] [' index] [' instrs]})) [' l] (
           d 0) [nil]}))
```

Listing 20: test5.l2

```
[printf (" This is how the quote macro is used. Now
    printing number in speechmarks "%i") (d 123)]
```

Listing 21: shell

```
./bin/l2evaluate bin/i386.a - abbreviations.l2
    comments.l2 - numbers.l2 - backquote.l2 -
    characters.l2 reverse.l2 strings.l2 - test5.l2
```

6.6 Conditional Compilation

Up till now, references to functions defined elsewhere have been the only things used as the first subexpression of an expression. Sometimes, however, the clarity of the whole expression can be improved by inlining the function. The following code proves this in the context of conditional compilation.

Listing 22: test6.l2

```
((if [> (d 10) (d 20)] fst frst)
  [printf (" I am not compiled!)]
  [printf (" I am the one compiled!)])
```

Listing 23: shell

```
./bin/l2evaluate bin/i386.a - abbreviations.l2
    comments.l2 - numbers.l2 - backquote.l2 reverse.l2
     - characters.l2 strings.l2 - test6.l2
```

6.7 Variable Binding

Variable binding is enabled by the continuation expression. continuation is special because, like function, it allows references 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 return
  {(continuation templet0 (params ...)
    {return expr0}) vals ...})
```

It is implemented and used as follows:

Listing 24: let.l2

```
(** Returns a list with mapper applied to each
    element.
(function map (l mapper)
  (with return
    {(continuation aux (in out)
      (if [nil? [' in]]
        {return [reverse [' out]]}
        {aux [rst [' in]] [lst [[' mapper] [fst [' in
            ]]] [' out]]})) [' l] [nil]})))

(function let (l)
  ('(with return
    (,[llst (' jump) ('(continuation templet0
      (,[map [fst [' l]] fst])
      {return (,[frst [' l]])})) [map [fst [' l]]
          frst]]))))
```

Listing 25: test7.l2

```
(let ((x (d 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 reference shadowing, so let expressions can be nested without worrying, for instance, about the impact of an inner templet0 on an outer one.

Listing 26: shell

```
./bin/l2evaluate bin/i386.a - abbreviations.l2
    comments.l2 - numbers.l2 - backquote.l2 -
    characters.l2 strings.l2 let.l2 - test7.l2
```

6.8 Switch Expression

Now we will implement a variant of the switch statement that is parameterized by an equality predicate. The switch selection function implements 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:

Listing 27: switch.l2

```
(function switch (l)
  (`(let ((tempeq0 (,[fst [' l]])) (tempval0 (,[frst
      [' l]])))
    (,(with return
      {(continuation aux (remaining else-clause)
        (if [nil? [' remaining]]
          {return [' else-clause]}
          {aux [rst [' remaining]]
            (`(if (,[llllst (` invoke) (` [' tempeq0
                ]) (` [' tempval0]) [ffst [' remaining
                ]] [nil]])
              (,[frfst [' remaining]]) (,[' else-
                clause]))))}))
        [rst [reverse [rrst [' l]]]] [fst [reverse ['
            l]]]})))))
```

Listing 28: test8.l2

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

Listing 29: shell

```
./bin/l2evaluate bin/i386.a - abbreviations.l2
    comments.l2 - numbers.l2 - backquote.l2 reverse.l2
     - switch.l2 characters.l2 strings.l2 let.l2 -
    test8.l2
```

6.9 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:

```
(environment env0 (args ...) expr0)
->
(function env0 (cont0 args ...)
  {[' cont0] expr0})

(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:

Listing 30: closures.l2

```
(function environment (l)
  (`(function (,[fst [' l]]) (,[lst (` cont0) [frst
      [' l]]])
    {[' cont0] (,[frrst [' l]])})))

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

(function ; (l)
  (`(with return (,[lllst (` invoke) [fst [' l]] (`
      return) [rst [' l]]]))))

(function : (l)
  (`(with return (,[lllst (` jump) [fst [' l]] (`
      return) [rst [' l]]]))))
```

Listing 31: test9.l2

```
(environment adder (x)
  (lambda (y) [+ [' x] [' y]]))

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

Listing 32: shell

```
./bin/l2evaluate bin/i386.a - abbreviations.l2
    comments.l2 - numbers.l2 - backquote.l2 reverse.l2
     - characters.l2 strings.l2 closures.l2 let.l2 -
    test9.l2
```

6.10 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:

Listing 33: assume.l2

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

Listing 34: test10.l2

```
(function foo (x y)
  (assume [not [= [' x] [' y]]] (begin
    [set-char [' x] (char A)]
    [set-char [' y] (char B)]
    [printf (" %c) [get-char [' 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 [get-char [' x]], a load from memory, with (char A), a constant.

Listing 35: shell

```
./bin/l2evaluate bin/i386.a - abbreviations.l2
    comments.l2 - numbers.l2 - backquote.l2 reverse.l2
     - characters.l2 strings.l2 assume.l2 - test10.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.

7 Compilation Library

My L2 system provides a library called l2compile.a to enable the compilation of L2 source files. Below is a description of the functions this library exports; they can be invoked using any source language so long as the calling convention outlined above is adhered to. In what follows, I invoke these functions is from within the evaluator using the command: ./bin/l2evaluate ./bin/i386.a ./bin/l2compile.a - abbreviations.l2 comments.l2 - numbers.l2 - backquote.l2 reverse.l2 - characters.l2 strings.l2 closures.l2 let.l2 -. In fact, this command is so convenient that I have packaged it in a script called l2evaluate-with-compile at the root of the repository.

7.1 [library x]

x must be the path of an L2 source file.

Evaluates to the path of a static library produced from compiling the source file x.

Typing [puts [library (" test9.l2)]], pressing Enter, then pressing Ctrl-D prints ".libvORuaz.a".

7.2 [sequence x y]

x and y must be paths to static libraries.

Evaluates to the path of a static library whose's ordered sequence of object files is the concatenation of those of x and those of y.

Typing [puts [sequence (" bin/i386.a) [library (" test9.l2)]]] , pressing Enter, then pressing Ctrl-D prints ".libnApfuA.a".

7.3 [executable x]

x must be the path to a static library.

Evaluates to the path of an executable whose's behavior is the "concatenation" of the ordered object files in x.

Typing [puts [executable [sequence (" bin/i386.a) [library (" test9.l2)]]]] , pressing Enter, then Ctrl-D prints ".exe6u1i1s". Running this executable should give the same output as in the section above.

7.4 [copy x y]

x must be a path to a non-existing file. y must be a path to an existing file.

Copies the file at path y to the path x. Invokation evaluates to x.

Typing [puts [copy (" test9) [executable [sequence (" bin/i386.a) [library (" test9.l2)]]]]] , pressing Enter, then Ctrl-D prints "test9". Running this executable should give the same output as above.

7.5 [nil−library]

Evaluates to the path of an empty static library.

This function plays the role of "zero" in the arithmetic of static libraries.

Typing [puts [executable [nil−library]]] , pressing Enter, then Ctrl-D prints ".exeOj671O". Running this executable should do nothing.

7.6 [skip x y]

x must be a path to a static library. y must be a symbol not occuring in the static library x.

Evaluates to the path of a static library whose's first object file comprises a jump to y, whose's following object files are those of x in order, and whose's last object file comprises the label y.

This function is useful for jumping over static libraries produced by C compilers. Otherwise L2's top-down mode of execution would cause C function definitions to be executed upon encounter rather than only on invocation.

Typing [puts [executable [skip [sequence (" bin/i386.a) [library (" test9.l2)]] (" skipper)]]] , pressing Enter, then Ctrl-D prints ".exeBokO5Q". Running this executable should do nothing.

7.7 [concatenate x y]

x and y must be paths to L2 source files.

Evaluates to the path of an L2 source file produced by concatenating the source files x and y.

Typing [puts [concatenate (" abbreviations.l2) (" comments.l2)]], pressing Enter, then pressing Ctrl-D prints ".catnX1RiH".

7.8 [nil−source]

Evaluates to the path of an empty source file.

This function plays the role of "zero" in the arithmetic of source files.

Typing [puts [executable [library [nil−source]]]] , pressing Enter, then Ctrl-D prints ".exelbIIhj". Running this executable should do nothing.

7.9 [load x]

x must be the path to a static library.

Augments the current environment with the static library x. Evaluates to a handle to the loaded library.

Typing [load [library (" file1 .l2)]] , pressing Enter, then Ctrl-D should print "a". Entering (foo this text does not matter), pressing Enter, then Ctrl-D should print "b".

7.10 [unload x]

x should be the handle to a loaded library.

Removes from the current environment the loaded library to which x refers. Evaluates to an unspecified value.

In an environment where load has not yet ben used, type [unload [load [library (" file1 .l2)]]] , press Enter, then Ctrl-D. Entering (foo this text does not matter), pressing Enter, then Ctrl-D should produce an error.

7.11 [dynamic x]

x must be the path of a static library.

Evaluates to the path of a dynamic library that executes x on loading and that exports all the functions of x.

Typing [puts [dynamic [library (" comments.l2)]]], pressing Enter, then pressing Ctrl-D prints "./.libLDsGsm.so". The output of objdump −T ./.libLDsGsm.so should show that ∗∗ is an exported symbol.