Thoughts about a Lisp-to-C Compiler

Gene Michael Stover

created 23 March 2004
updated 15 July 2004

Contents

Chapter 1

Introduction

These are notes about a Lisp-to-C compiler. This document is not yet ready for publish consumption (or comment).

Chapter 2

Issues

function calls In progress as of about 8 July 2004. Should finish function calls, including testing, weekend of 17 or 18 July 2004.

memory management

translation overview

foreign functions

symbol table

symbol value

special forms block, return-from

- **flet**
- **labels**
- **let**
- **progn**
- **setq**
- **tagbody, go**

functions

macros

C run time apply

- **funcall**

type system

Chapter 3

Goals

I want a compiler that eats Lisp code & excretes C code. Goals for the C code, beginning with the most important, include:

1. correctly implements the Lisp code from which it was derived,
2. is portable across C compilers & operating systems (assuming that what the Lisp source code did was portable),
3. is straightforward, hopefully simple, so that a single programmer can create, verify, & debug the compiler itself,
4. some other goal,
5. some other goal,
6. several other goals,
7. be efficient at run-time.

Chapter 4

Memory Management

Rely on the C library to manage memory. In other words, the underlying C library performs garbage collection. We'll allocate memory with a function called `xmalloc`, which I write, & we won't free it.

The `xmalloc` function will probably sit on top of the `GC_malloc` function of the Boehm collector ([Boe,)].

Chapter 5

Values

How to implement Values?

Should every function or form return a Values instead of a single Lisp object? If Values is a Lisp object, No, but if Values is just a C array of Lisp objects, maybe.

Is there a way to implement Values so that it doesn't impact code that ignores Values (which is most code)?

What if the primary return value is returned by the C function but Values are stored in a Values Register? Code that doesn't care about Values always uses the C function return value, but Values-aware code can look in the Values Register. When Lisp returns a Values, it stuffs all of them into the Values Register & returns the first one from C. That means C functions should clear the Values Register when they return, even if they are otherwise unaware of Values.

Wait, No! If every C function cleared the Values, then youcouldn't return a Values from a function you called. So the Values Register is normally clear. It's set by VALUES, access & cleared by MULTIPLE-VALUE... functions, & printed & cleared by top-level.

Will this work?

This should work:

```
(multiple-value-bind (a b c) (values 1 2 3))
```

because the VALUES sets the Values Register & returns 1. MULTIPLE-VALUE-BIND ignores the 1 but retrieves the Value Register & clears it.

How about this (at top level):

```
(if (zerop (values 0 1))
    1
   2)
```

Unless the Values Register is cleared as or when we call ZEROP, the top-level printer will think there are two values & print them. So the Values Register must be cleared for every function call. How about for special forms & macros?

Yuck.

What about the Values as Lisp Object technique? Forms which are not Values-aware return a regular Lisp object. Values returns a Values object. Problem with this technique is that we must scrube every value before using it as an argument to a function. It's effectively the same as the Values Registry technique.

The best technique might be to always return a Values, but implement it as a C array of pointers, terminated by NULL. Most expressions return an array of two elements, second is NULL. When using the return value in a Values-unaware way, always use the first array element. Values-aware uses might access all elements. If a C function returns NULL, it's a serious run-time error, probably out-of-memory or "Crisp's programmer seriously screwed up".

Caller must not alter or reuse the Values C array; that allows called function to return a constant, statically allocated array if appropriate.

Most forms will return a Values C array of two elements. The first element is (a pointer to) the only return value. The second element is NULL.

I can use a C run-time convenience function to return the Values C array.

A VALUES form compiles like this:

```
(values a b c)
```

compiles to

```
/* Value-of symbol A compiles to */
extern Object **M456 (/* lexy, dyna */);
/* Value-of symbol B compiles to */
extern Object **M457 (/* lexy, dyna */);
/* Value-of symbol C compiles to */
extern Object **M458 (/* lexy, dyna */);

/* (values a b c) compiles to */
Object **
M459 (lexy, dyna)
{
  /* List of forms to eval first */
  static Object **(*forms[]) (/* lexy, dyna */ = {
    &M456, &M457, &M458 };

  return CRISPY_Values (forms, sizeof forms / sizeof forms[0],
                        lexy, dyna);
}
```

The run-time library contains the `CRISPY_Values` function. For the most part, it evaluates the forms & stowes their values in its C array, but if there is an exception, `goto`, `return-from`, or other control-changing form, it stops the normal processing & returns that special control-changing object immediately.

```
/* Pseudo-C, first attempt */
Object **
CRISPY_Values (forms, len, lexy, dyna)
     Object **(*forms[]) (/* lexy, dyna */);
     int len;
{
  Object **rc, **tmp;
  int i = 0;

  rc = (Object **) xmalloc ((len + 1) * sizeof *rc);
  if (len > 0) {
    do {
      tmp = (*forms[i]) (lexy, dyna);
      rc[i] = tmp[0];
    } while (i < len && !IsControlChange (rc[i - 1]));
    if (IsControlChange (tmp[0])) {
      rc[0] = tmp[0];
      i = 1;
    }
  }
  rc[i] = NULL;
  return rc;
}
```

Rewrite. Can optimize for len = 0. Try to improve the loop, too.

```
Object **
Values (forms, len, lexy, dyna)
     Object **(forms[]) (/* lexy, dyna */);
     int len;
     ??? lexy;
     ??? dyna;
{
  Object **rc, **tmp;
  int i;
  static Object *empty[] = { NULL };

  if (len == 0) {
    rc = empty;
  } else {
    rc = xmalloc ((len + 1) * sizeof *rc);
    i = 0;
    do {
      tmp = (*forms[i]) (lexy, dyna);
      rc[i] = tmp[0];
    } while (!IsControlChange (tmp[0]) && ++i < len);
```

```
    if (IsControlChange (rc[i])) {
      /* One of the forms tossed an exception, executed a
       * GO to, or a RETURN.  We must copy that value to
       * the front of the array.  And remember to append
       * a NULL. */
      rc[0] = rc[i];
      rc[1] = NULL;
    } else {
      /* Normal control.  We evaluated all the forms. */
      assert (i == len);
      assert (rc[i] == NULL);
    }
  }
  return rc;
}
```

Chapter 6

Block, Return, & Return-From

RETURN is easy. It can be a Lisp macro:

```
(defmacro return (x)
  `(return-from nil ,x))
```

BLOCK puts a Block object on the lexical context. At compile-time, the Block object needs the BLOCK's label. At run-time, the label is not needed, but it might be useful for debugging.

At compile-time, must search the lexical environment to find the Block. At run-time, we don't search at all; the Block's position is a known number of items from the top of th elexical environment.

The RETURN-FROM creates a ReturnFrom object, puts the Block object's id (address) on it, & returns. It gets the Block object's id by searching the lexical environment, but that search is just a numeric iteration from the top of the lexical stack.

The compiler finds the Block object in the lexical. RETURN FROM NIL is a special ase; the compiler finds the Block nearest the top of the stack, regardless of the label.

Chapter 7

Functions

You should be able to RETURN-FROM or RETURN from a named function as if it were a BLOCK. It might be easy to implement this with the Block run-time function, like this.

```
/* Pseudo-C for a compiled Lisp function */
Object **
ALispFunc (args, lexy, dyna)
   Object *args[];
   ??? lexy;
   ??? dyna;
{
  PrepareFuncArgs (args, &lexy, &dyna);
  return Block (&mySymbol, lexy, dyna);
}
```

where "mySymbol" is the name of the C Symbol object that will label the block.

Maybe it's better to use a macro implementation. "DEFUN MOO (X) X)" WOULD EXPAND TO:

```
(cybertiggyr-crisp-set-symbol-function
  'moo
  #'(lambda (x)
      (block moo
        x)))
```

WHICH WOULD COMPILE TO

```
Object M001; /* Symbol X, initialized elsewhere */

/* C func which returns the value of X when X is top
 * of the lexical stack. */
Object **
```

```
M000 (lexy, dyna)
{
  Object **rc;

  rc = xmalloc (2 * sizeof *rc);

  /* Some validity checks the compiler could insert. */
  assert (lexy != NULL);
  assert (lexy->aa != NULL);
  assert (lexy->aa->type == LispType_Symbol);
  assert (strcmp (lexy->aa->u.symbol.name, "X") == 0);
  assert (lexy->bb != NULL);

  /* Retrieve the value bound to X & store it in our
   * Values which we'll return. */
  rc[0] = lexy->bb;
  rc[1] = NULL;
  return rc;
}

Object M000; /* symbol M00 */

/* (block moo x) */
Object **
M003 (lexy, dyna) {
  static Object **(*forms[]) (/* lexy, dyna */) = {
    &M000 };
  return Block (forms, sizeof forms / sizeof forms[0],
                lexy, dyna);
}

/* (function (lambda (x) x)) */
Object **
M004 (lexy, dyna) {
... ugh ...
}
```

I see that I must think about what "(#'λ (...) ...)" compiles to.

For one, it compiles to a Lisp Closure object. The Closure must track:

- the lexical environment in which it was defined,
- pointer to the C function,
- lambda list, or a C function or data structure that binds the actual arguments to the format arguments.

I can implement funcall as a Lisp function, so there is no need to put it in the C run time library or the compiler. That funcall is:

```
(defun funcall (fn &rest args)
  (apply fn args))
```

For efficiency, I might want to implement funcall in the compiler, but that can wait.

So how about apply? I think it must be in the run time library. Call it `Apply`. It needs to actual arguments, the lexical & dynamic environments, & the closure.

It needs the actual arguments, obviously.

It needs the dynamic environment because the closure will need it.

Does it need the lexical environment? Ah, yes, because... well, maybe not. It might have a *lexy* formal argument because all the functions do.

It needs the closure because that tells Apply how to bind the actual to the formal arguments; the closure saves the lexical environment & the closure has the C function that does the work.

7.1 Pseudocode, First Try

```
Object **
Apply (args, len, closure, lexy, dyna)
   Object *args[];
   int len;
   Object *closure;
   ??? lexy;
   ??? dyna;
{
  Bind the actual args to the format args.
  Save as lexy2 & dyna2.

  /* Call the C function. */
  return (*closure->u.closure.fn) (lexy2, dyna2);
}
```

We don't want `Apply` to figure out which args must be bound to lexical & special symbols.

The compiler can figure this out when it compiles the function, emitting the Closure object & C functions.

How would it work if I stored the lambda list verbatim with notes about which formal arguments are lexical & special? (The compiler can figure out which are special & lexical.)

fixme: This was a second(?!) chapter named Functions, I guess. Need to merge them.

A Lisp function compiles to multiple C functions & objects.

At the lowest level, it compiles to a C function with a fixed number of named arguments. That C function does the real work.

At a layer above the low-level C function, there's a high-level C function. The Lisp actual arguments are delivered to the high-level C function in an array of pointers to Object. The second argument to that C function is the length of the array. The third argument to that function is the context. The high-levle C function parses the actual arguments into the formal arguments, including `&optional`, keyword & other arguments. It creates a new context (probably by calling `Let`), & it calls the low-level C function.

At a layer even above the high-level C function is the Closure. It's a data object that has a pointer to the high-level C function & the context in which the Lisp function was created.

Here's an example.

This Lisp form:

```
(defun moo (x y) (+ x y))
```

could compile to this:

```
/* Assume these C Symbol objects */
Symbol symX = { ..., "X", ... };
Symbol symY = { ..., "Y", ... };
Symbol symPlus = { ..., "+", ... };
Symbol symMoo = { ..., "MOO", ... };

/* Function that evalutes symbol X */
Values *
L1 (Context *ctx) {
  return GetSymbolValue (&symX, ctx);
}

/* Function that evaluates Y */
Values *
L2 (Context *ctx) {
  return GetSymbolValue (&symY, ctx);
}

/* Function that evaluates (+ x y) */
Values *
L3 (Context *ctx) {
  Closure *closure;
```

```
  Object **args;

  closure = GetFdefinition (&symPlus, ctx);
  args = (Object **) xmalloc (2 * sizeof *args);
  args[0] = L1(ctx)->value[0];
  args[1] = L2(ctx)->value[1];
  return (*closure->fn) (args, 2, ctx);
}

/* Low-level C function that gets bound to MOO */
Values *
LowLevel123 (Object *x, Object *y, Context *ctx) {
  return L3 (ctx);
}

/* High-level C function for MOO */
Values *
HighLevel124 (args, len, ctx)
  Object *args[];
  int len;
  Context *ctx;
{
  /* ??? fixme ???
   * To be continued after I design Let */
}
```

Chapter 8

Structure of the C Program

Crisp compiles Common Lisp programs to C programs. This chapter discusses the architecture & details of the C programs.

There are excerpts from my pen-&-paper notebook. I wrote so much that I don't have the time to transcribe it all, which is a bummer.

I guess I'll have to implement an `eval` function for Crisp. It will interpret the Lisp source code that it's compling, & at the same time, it will emit C code that would perform an equivalent evaluation once it is compiled & run. The compiler *must* implement its own `eval` that evaluates the source code as it's being compiled, & also compiles it to C as it's being evaluated. There's no way around it. (I wrote a lot about it in my notebook.)

The compiler compiles one top-level form at a time. At that level of abstraction, the operation of the compiler is as in Figure 8.1.

8.1 Forms become functions

Each Lisp form in the source code becomes a function in the C code. Even fetching the value bound to a single symbol becomes a

- Until end-of-input
  - Read the next top-level Lisp form.
  - Compile & evaluate the form.
- Finish emitting code (if necessary), or cleanup.
- Exit.

Figure 8.1: Pseudocode for the Lisp-to-C compiler at top-level

C function. It might not be the most efficient way for the program to work at run-time, but it's simple.

In other words, assuming `x` is a symbol that is bound to some value, then this form: "`x`" compiles to something like this:

```
/* Earlier in the program, the X symbol was internned
 * like this. */
Symbol L2 = { ... "X", ... };

/* Then the form in question is compiled */
Values *
L123 (Environment *environ) {
  return GetSymbolValue (&L2, environ);
}
```

This example assumes that the C function `GetSymbolValue` takes a C symbol object & an execution environment. It figures out what value is bound to the symbol & returns a pointer to a dynamically allocated Values object that contains the value bound to the symbol.

If we were to compile an even simpler form, such as the integer 2, we might get this C function:

```
/* The first time the integer 2 was encountered
 * by READ, it was "interned", in a way, even
 * thought it's not a symbol. */
Integer L5 = { ..., 2, ... };

/* When we need the value for "2", it compiles to
 * this C function (or something like it) */
Values *
L4232 (Environment *environ) {
  /* Unlike the value bound to a symbol, the value
   * for an integer never changes, so we don't need
   * to look it up each time. */
  /* We might get a small optimization by statically
   * allocating the Values. */
  static Values values = { ..., &L5, ... };

  return &values;
}
```

Notice that we don't need to emit a new function each time we encounter the same symbol or the same number. The compiler's symbol table can match symbols & other objects with the names of the C functions which return their values. It works for symbols, numbers, other constants, & every other type of Lisp expression.

For example, the Lisp expression `(moo 1 2)`, which calls the function bound to `moo`, might compile to this:

```
/* The symbol table was initialized to have the C
 * object for NIL */
Symbol Nil = { ..., "NIL", ... };

/* When it was read the first time, the symbol MOO
 * interred as this */
Symbol L2000 = { ... "MOO", ... };

/* The integers 1 & 2 interred like these */
Integer L2010 = { ..., 1, ... };
Integer L2011 = { ..., 2, ... };

/* The function which returns the value for 1 */
Values *
L2057 (Environment *environ) { ... };

/* The function which returns the value for 2 */
Values *
L2062 (Environment *environ) { ... };

/* Later, the expression (moo 1 2) itself
 * compiles to */
Values *
L3113 (Environment *environ) {
  Closure *closure;
  Object **args; /* dynamically allocated array of args */

  closure = GetFdefinition (&L2000, environ);

  /* Allocate the array from the heap just in case
   * some other function still refers to it after this
   * function returns.  I'm not sure it's necessary,
   * but it's a safety feature.  It'll have a run-time
   * cost, so if it turns out not to be necessary,
   * it'd be a good idea to put args[] on the C
   * stack. */
  args = (Object **) xmalloc (2 * sizeof *args);
  args[0] = L2057 ();
  args[1] = L2062 ();
  return (*closure->fn) (args, 2, environ);
}
```

It sure would be cool to send the arguments on the C function call stack. That would require variable arguments. You can do

them in C, but it's not all that portable, & I'm not sure it'd be much of an improvement. I'll look into doing it for a later version, though. It might be worthwhile.

This begs the question: How to implement the `apply` Lisp function? See Section 9.1.

Here's an example with nested expressions. We'll compile "`(1+ (+ 1 2))`".

```
/* The integers 1 & 2 interred like these */
Integer L2010 = { ..., 1, ... };
Integer L2011 = { ..., 2, ... };

/* The symbol 1+ */
Symbol L2035 = { ..., "1+", ... };

/* The symbol + */
Symbol L2036 = { ..., "+", ... };

/* The function which returns the value for 1 */
Values *
L2057 (Environment *environ) { ... };

/* The function which returns the value for 2 */
Values *
L2062 (Environment *environ) { ... };

/* The function which evaluates (+ 1 2) */
Values *
L3011 (Environment *environ) {
  Closure *closure;
  Object **args;

  closure = GetFdefinition (&L2036, environ);
  args = (Object **) xmalloc (2 * sizeof *args);
  args[0] = L2057 (environ);
  args[1] = L2062 (environ);
  return (*closure->fn) (args, 2, environ);
}

/* The function which evaluates (1+ (+ 1 2)) */
Values *
L3087 (Environment *environ) {
  Closure *closure;
  Object **args;

  closure = GetFdefinition (&L2035, environ);
```

symbol The lexical symbol.

value The value bound to the symbol.

type Binding type: value, function, macro, label.

next The next Binding in the list. This might not be an intrusive field; we could use a Lisp list.

Figure 8.2: The fields of a lexical Binding record.

```
  args = (Object **) xmalloc (1 * sizeof *args);
  args[0] = L3011 (environ);
  return (*closure->fn) (args, 1, environ);
}
```

8.2 Lexical Context

Lexical variables can be expressed in a linked list. In Pascal with its nested procedures, the nodes of the list are on the stack, but in Lisp they have indefinite extent.

The obvious way is for each activation record to create a Frame in the list, but I *think* each lexical variable can become an element in the list. If so, the elements in the lexical list are Bindings, shown in Figure 8.2.

At run time, we only need the *value* of the Binding, but the other fields could be useful for debugging or for automatic consistency checking at run time.

The compiler tracks the lexical environment. If the code refers to a lexical symbol, the compiler finds the symbol int he lexical environment. That location will be the same at run time, so no need to do the search at run time. The compiler can output code to fetch the binding without conditionals. That code becomes a bunch of indirections up the linked list.

For example, it we compile the Lisp symbol "X" (in other words, a Lisp expression to return the value of the symbol X) in the lexical environment ((x "hi") ...), the resulting C code would be "`ctx->value`".

Another example: Compile "Y" in the lexical context ((x a) (z nil) (y 2) ...). It produces the C code "`ctx->ctx->ctx->value`".

The lexical environment is altered by let, labels, flet, macrolet. Also, labels for go or return-from are lexical, so the forms which create them also alter the lexical environment. Purely dynamic labels, such as for catch / throw, do not belong in the lexical environment.

At compile time, each crisp-eval-* function has lexical context argument & dynamic context argument.

Compiling let: Compiler evals the initializers into temps (prolly an array in the C code). Then for each symbol, if it's lexical, make a Binding for it & push onto the lexical environ. Otherwise, it's global so push value onto the symbol's *specials* field.

No. That's wrong. At compile time, check a symbol to see which of these four cases apply:

1. Symbol is neither special nor lexical. So it gets a new lexical record.
2. Symbol is already in lexical environment but not special. So it gets a new lexical record.
3. Symbol is special but not lexical. So it gets a new dynamic record (push onto symbol's *specials*).
4. Symbol is both special & lexical. This is a compile-time error.

To compile setq, look for the symbol in the lexical environment. At compile time, you alter the binding itself. You output C code to do that at run time, like this: "`ctx->ctx->ctx->ctx->value =`", followed by the new value, which was produced by calling a C function & storing its return value in an array of temporaries.

If the target symbol for setq is special, just assign the new value to the top entry of that symbol's *specials* stack.

It is a compile time error if the target symbol is special & is in the lexical environment.

8.3 Dynamic Context

The C Symbol data type will have a field for special variables. That field is a stack of pointers to Objects. Actually, it's a pointer to a stack. If the field is null, the symbol is not special & any time we try to fetch its value, it had better be in the lexical context. If the stack exists, the symbol is special.

The C implementation of setq stuffs a new value into the top of the stack. It does not *push* a new value; it replaces the existing top of the stack, unless the stack is empty.

The C implementation of let pushes a new value onto the Symbol's stack if the Symbol is special. That determination is made at compile-time.

If the Symbol's stack exists but is empty, the symbol is special but unbound. Can this situation exist, or is a symbol automagically bound to something (probably nil) when I make it special? It's bound to nil when I make it special with defvar & do not supply

an initial value. Can't you remove a symbol's special values with make-unbound or a function with some similar name? Sounds like it is possible for a special symbol in Lisp to be unbound. I'll represent that situation in the C program as a Symbol with an existing but empty stack of special values.

A quick & easy way to implement setq is to pop the special stack, then push a value. This works if popping an empty stack is not an error. So when I create the "stack of pointers to Objects" C module, I should make sure that popping an empty stack is not an error. Maybe it can return null. These stacks will not be directly accessible from Lisp; they are part of the C implementation.

8.4 Catch & Throw

A Lisp catch form compiles to a C function which alters the dynamic context. In a stripped-down implementation, I think it would not need to alter any data at all, but for debugging purposes, the dynamic context should probably be held explicitly in a stack of symbols & other things which are active. (Values bound to special symbols are not in this stack; they are in stacks within the C Symbol objects themselves.)

A Lisp throw compiles to a C function which returns a C Exception object. All C functions in the call stack will notice that the C functions they called return Exception objects, & they will propagate them by returning them without modification.

Every catch form (as a C function) will examine the return value of whatever C functions it called. If one of them returns an Exception on the same symbol the catch defines, the catch form will intercept it & take action. Otherwise, the catch returns the value if it's an exception, error, or goto, or it continues calling the forms within it.

If an exception arrives at the top-level loop by being returned from a C function, it's an error. The C program exits with an error message where an interactive Lisp system would probably enter the debugger loop.

Notice that there is no place in the C program where the catch or throw require an examination of the entire dynamic context. Instead, they & other forms cooperate to figure out how to handle the exception. So there is no need to keep the entire dynamic context in an explicit data structure, but it might be useful for debugging. If the C program retained a list of the labels in active catch forms, a throw form could examine the list & print a helpful error message if the exception it is about to throw will not be caught.

On second thought, a helpful error message does not require

a data structure of the active catch labels. The throw could stuff error information into the Exception object along with the symbol it's throwing. If the Exception arrives at the top-level loop, the information in it can be used when printing the error message.

So there is no need to keep an explicit data structure that lists the symbols active in catch forms.

Notice that go & return-from will require some type of data structure so they can ensure that only a lexically nested form will handle their special C return values. Otherwise, a dynamically nested form might handle their values.

Chapter 9

C Runtime Library

9.1 apply

Apply is what I'd guess you'd call a built-in Lisp function. It's C version has these arguments: array of pointers to actual arguments, length of the array (i.e., number of arguments), & the environment.

The first element in the array is a closure or a symbol that is bound to a closure.

The high-level C function is:

```
Values *
Apply (args, len, environ)
     Object *args[];
     int len;
     Environment *environ;
{
  Closure *closure;
  Values *values;
  Object **args2;

  if (IsClosure (args[0], environ)) {
    closure = (Closure *) args[0];
  } else if (IsSymbol (args[0], environ)) {
    closure = GetFdefinition ((Symbol *) args[0], environ);
    /* Ensure that the symbol is bound to a function.
     * In the future, do a proper check, print a
     * helpful error message if necessary, & abort. */
    assert (closure != NULL);
  } else {
    /* It's not a closure, & it's not a symbol.  That's
     * an error.  In the future, print something helpful
```

```
     * before bailing out.   Also, bail-out in a well
     * behaved way. */
     abort ();
  }

  /* Now we call Append on the rest of the arguments.
   * Not yet sure how that happens, what comes
   * back.  So this is an approximation. */
  values = Append (args + 1, len - 1, environ);

  /* Should check 'values' for errors here. */

  args2 = values->obj[0];
  return (*closure->fn) (args2, length of args2, environ);
}
```

9.2 IsSpecial

This function accepts any Lisp object as its argument. If the object is a symbol & is special, it returns true.

```
Boolean
IsSpecial (Object *x, Context *ctx) {
  /* It's special if it's a symbol & its stack for
   * special bindings is allocated. */
  return x->type == LispType_Symbol && x->u.symbol.stack != NULL;
}
```

9.3 let

Let creates a new context, evaluates a Lisp expression (as C code), & then restores the old context. Let is pretty easy to implement in C because every Lisp expression compiles to a C function.

```
Object *
LowLevelLet (Binding lexy[], int lexy_len,
             Binding dyna[], int dyna_len,
             Object *(*expr) (Context *ctx),
             Context *ctx)  /* lexical context */
{
  Object *values;
  Context *ctx2;
  int i;

  ctx2 = CONTEXT_New (ctx);
```

```
  for (i = 0; i < lexy_len; ++i) {
    assert (! IsSpecial (lexy[i].symbol));
    HT_Insert (ctx2->ht, lexy[i].symbol, lexy[i].value);
  }
  for (i = 0; i < dyna_len; ++i) {
    assert (IsSpecial (dyna[i].symbol));
    STACK_Push (dyna[i].symbol->special, dyna[i].value);
  }
  values = (*expr) (ctx2);
  /* Un-do the specials */
  for (i = 0; i < dyna_len; ++i) {
    assert (IsSpecial (dyna[i].symbol));
    STACK_Pop (dyna[i].symbol->special);
  }
  return values;
}
```

9.4 progn

`Progn` can be implemented as a macro.

9.5 Stack of Lisp Objects

9.6 object-stack

This is an entire, small C module, actually.

We can make a stack, push `Object *` onto it, pop them from it, & examine the top of the stack.

Chapter 10

Examples

These are some examples of Lisp code & the C code to which they might compile.

10.1 Constant Integer

Here's a Lisp program containing just one top-level form, an integer:

```
;; This is a trivial Lisp program
3
```

The reader reads & discards the comment. Then the reader reads the "2" & sends it to the compiler's `eval`.

`Eval` recognizes it as an integer. That integer has not been internned, so get a new C symbol for it. That symbol is `L123` in this example.

To the "decls.i" output file, emit this C code:

```
extern Object L123; /* declare 2 */
```

To the "body.i" output file, emit this C code:

```
Object L123; /* object for 2 */
```

To the "init.i" output file, emit this C code:

```
  L123.type = LispType_Integer;
  L123.u.integer = 2;
```

That chunk of initialization code will be `#include`d into `main` where it will be executed early in the C program, before any Lisp forms are evaluated.

Now the compiler's `eval` needs a C function to evaluate this expression. There isn't one in the symbol table already, so it creates one. To the "decls.i" output file, it emits this C code:

```
Object *L124 (Context *ctx);
```

To the "exprs.i" output file, it emits this C code:

```
/* top level expression "2" */
Object *
L124 (Context *ctx)
{
  return GetSymbolValue (&L123, ctx);
}
```

To the "toplevel.i" output file, emit this C code:

```
  L124 (ctx);
```

Whew! That's a tiny Lisp program, but it generates a lot of C code.

The C skeleton file that we compile to produce the final program is like this (possibly omitting some details):

```
/* The C program skeleton */
#include standard C stuff
#include posix stuff

#include "decls.i"
#include "body.i"

int main (argc, argv)
     int argc;
     char *argv[];
{
  int rc = 0;
  Object *context_object;
  Context *ctx;

  context_object = AllocObject (LispType_Context);
  context_object.u.context.??? = ???;
  ctx = &context_object.u.context;

#include "init.i"
#include "toplevel.i"

  return rc == 0 ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Appendix A

Other File Formats

- This document is available in multi-file HTML format at http://lisp-p.org/crisp/.
- This document is available in Pointless Document Format at http://lisp-p.org/crisp/crisp.pdf.

Bibliography

[Boe] Hans-J. Boehm. A garbage collector for c & c++. http://www.hpl.hp.com/personal/Hans%5FBoehm/gc/.