Doing Evil Things with Common Lisp

Overview

- I use SBCL on a 32bit Intel Mac (OS X)
- Loading code that is normally a Linux driver into a Lisp image
- Creating CFFI bindings to foreign code
- Writing Lisp code to load binary files using the binary-types library
- Parsing Mach-O (OS X) executable files
- Using debug information to generate foreign bindings automatically

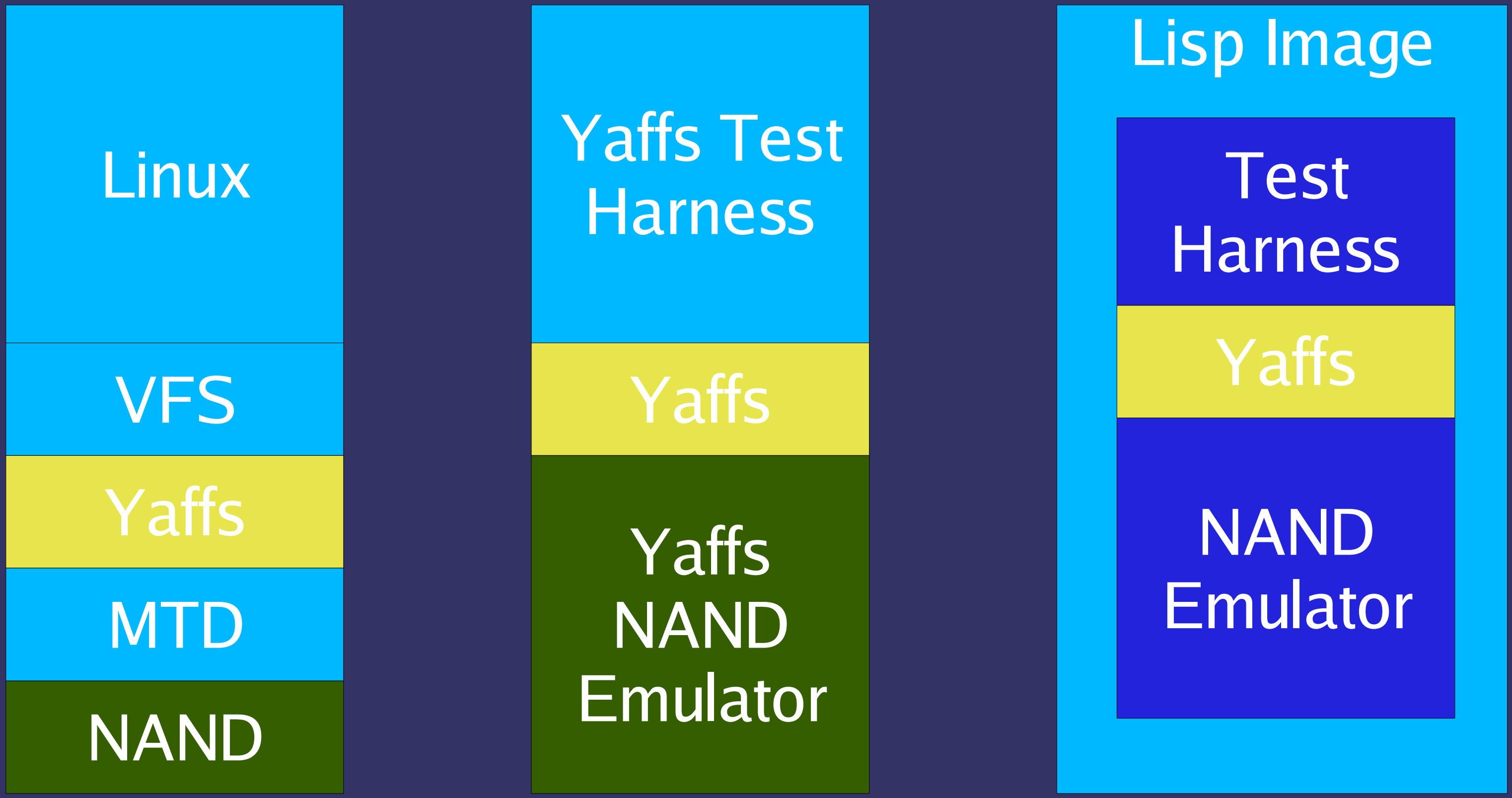
My Background

- 7 years of C/C++ experience doing embedded systems work
- 1 year of C/C++ in the games industry obsessing over performance and memory usage
- 2.5 years of Common Lisp with nothing of note to show for it & I still feel like a novice. But I'm having fun. Learn a language in 10 years and all that.

Project Background

- YAFFS is an Open Source filesystem driver, designed for NAND flash
- Runs under Linux & as a standalone application
- I know the author and he keeps bugging me to contribute again
- But I want to keep using Lisp, not C

Yaffs Software Stack



Typical Usage Stacks

What I want

Yaffs in Lisp

- ➲ Pretty easy to compile as a dynamic library & load within SBCL
  - Other Linux drivers might need a thin wrapper to provide link-time symbols
  - On OS X “gcc -bundle ...”
- ➲ Ideally I want to grow my Lisp image into a toolbench for working with Yaffs
  - Inspect internal Yaffs state
  - Call random functions
  - Add new functions/callbacks
  - Do all the things I enjoy doing in a pure Lisp en-vironment

Bindings

- Loading a DLL is the really easy bit
- Writing bindings to call that code is harder
- Doing it by hand is somewhere between tedious and hard
  - Yaffs has a bookkeeping struct that has about 220 members
- Doing it automatically doesn't always work out well
  - SWIG is something of a solution
    - Often needs to generate C/C++ code that needs to go into the original library
  - No maintained Lisp generators that I'm aware of

CFFI is Really Nice

-
  ```
  typedef struct foo {
          int a;
          float b;
  };
  ```

-
  ```
  int bar(float a){...}
  ```

-
  ```
  (defcstruct foo
          (a :int)
          (b :float))
  ```

-
  ```
  (defcfun bar :int
          (a :float))
  ```

Debuggers Are Your Friends

- C/C++ debuggers need to know a lot about the program they are debugging
  - Function signatures
  - Object layout and member names
  - How type defines and pointers map to other types
  - Sometimes even pre-processor macros
- They get this info by parsing special compiler generated debugging data
- The debug info is fairly easy to parse

The Idea

- Write a Lisp program to parse the debug data directly from a C/C++ binary that has been compiled with debugging turned on
- Automatically generate CFFI information that will be correct for that binary
  - No messing with pre-processor
- Usage
  1. Obtain binary with debugging info
  2. Generate bindings
  3. Woohoo
- Limitations
  - No way to get fully inlined functions
  - C++ name mangling still a per-compiler issue

A C/C++ Debugger written in CL??

- Totally pie in the sky right now
- But, how cool would that be?
- Not that much of a stretch, only need to be able to write to memory where the library is located to insert break points & then trap signals on the Lisp side
- Could use dirty tricks to replace the C functions with calls to Lisp functions
- Apart from some details, it's probably not that hard – certainly nothing “new”

Parsing Binary Files

- In C, you just create a struct of the correct layout, and use fread
- Common Lisp has no built in support for this idiom
- BINARY-TYPES is a Cliki library that lets you use the C idiom

-
  ```
  (define-binary-class mach-header ()
         ((magic          :binary-type 'u32)  ; should be #xfeedface
          (cpu-type       :binary-type 'u32)
          (cpu-subtype    :binary-type 'u32)
          (file-type      :binary-type 'mach-filetype-constants)
          (ncmds          :binary-type 'u32)
          (sizeof-cmds    :binary-type 'u32)
          (flags          :binary-type 'u32)))
  ```

-
  ```
  (with-open-file (stream “foo”)
         (read-binary 'mach-header stream))
  => CLOS Object of type MACH-HEADER
  ```

Mach O files

- Consists of
  - Header (previous slide)
  - Some number of load commands

    -
      ```
      (define-binary-class load-command ()
            ((cmd      :binary-type 'load-command-constants)
             (cmd-size :binary-type 'u32)))
      ```
  - Some number of sections containing real data
- Symbol table command looks like

  -
    ```
    (define-binary-class symtab-command () ; LC_SYMTAB
          ((command-header :binary-type 'load-command)
           (symoff         :binary-type 'u32)
           (nsyms          :binary-type 'u32)
           (stroff         :binary-type 'u32)
           (strsize        :binary-type 'u32)
           (symbols        :accessor symbols-of :initform nil)))
    ```
- Pretty much the same as ELF files

Stabs Debugging Format

- Simple string format
  “name:symbol-descriptor type-information”
- char:t(0,9)=r(0,9);0;127;
  typeid 't(0,9)', named char, with range 0..127
- :t(0,7)=*(0,9)
  typeid 't(0,7)' is a pointer to '(0,9)'
- myStruct:T(0,17)=s96a:(0,2),0,32;b:(0,2),32,32;c:(0,18),64,704;;
  's96' -> structure of size 96 bits
  'a:(0,2),0,32' -> name is 'a', type is '(0,2)', 0 bits offset from struct start, size 32 bits
- Way easier than parsing C/C++

C Code/CFFI Output

➲

```c
typedef int foo;
typedef char* string;
struct myStruct
{
    int a;
    int b;
    int c[22];
};

typedef struct
{
    int c;
} my2Struct;

typedef enum
{
    Efoo,
    Ebar
} foobar;

enum {
    foo2
} foobar2;

typedef union
{
    int i;
    char c;
} myunion;
```

➲

```lisp
(DEFCTYPE FOO INT)
(DEFCTYPE STRING CHAR*)
(DEFCSTRUCT MYSTRUCT
        (A INT OFFSET 0 COUNT 1)
        (B INT OFFSET 4 COUNT 1)
        (C INT OFFSET 8 COUNT 22))

(DEFCTYPE INT :int32)
(DEFCTYPE CHAR** :pointer)

(DEFCSTRUCT MY2STRUCT
        (C INT OFFSET 0 COUNT 1))


(DEFCENUM FOOBAR
        (EFOO 0)
        (EBAR 1))
(DEFCENUM ENUM-2003
        (FOO2 0))

(DEFCTYPE CHAR* :string)
(DEFCTYPE CHAR :char)
(DEFCTYPE FLOAT :float)
(DEFCTYPE DOUBLE :float)
(DEFCTYPE UNSIGNED-CHAR :uint8)

(DEFCUNION MYUNION
        (I INT COUNT 1)
        (C CHAR COUNT 1))
```

C Code/CFFI Output

➲
```
int func2(my2Struct
*sss) {};
```

➲
```
int main (int argc,
     char** argv){}
```

➲
```
int intfunc(int arg1){}
```

➲
```
(DEFCFUN func2 INT
     (SSS MY2STRUCT*))
```

➲
```
(DEFCFUN main INT
     (ARGC INT)
     (ARGV CHAR**))
```

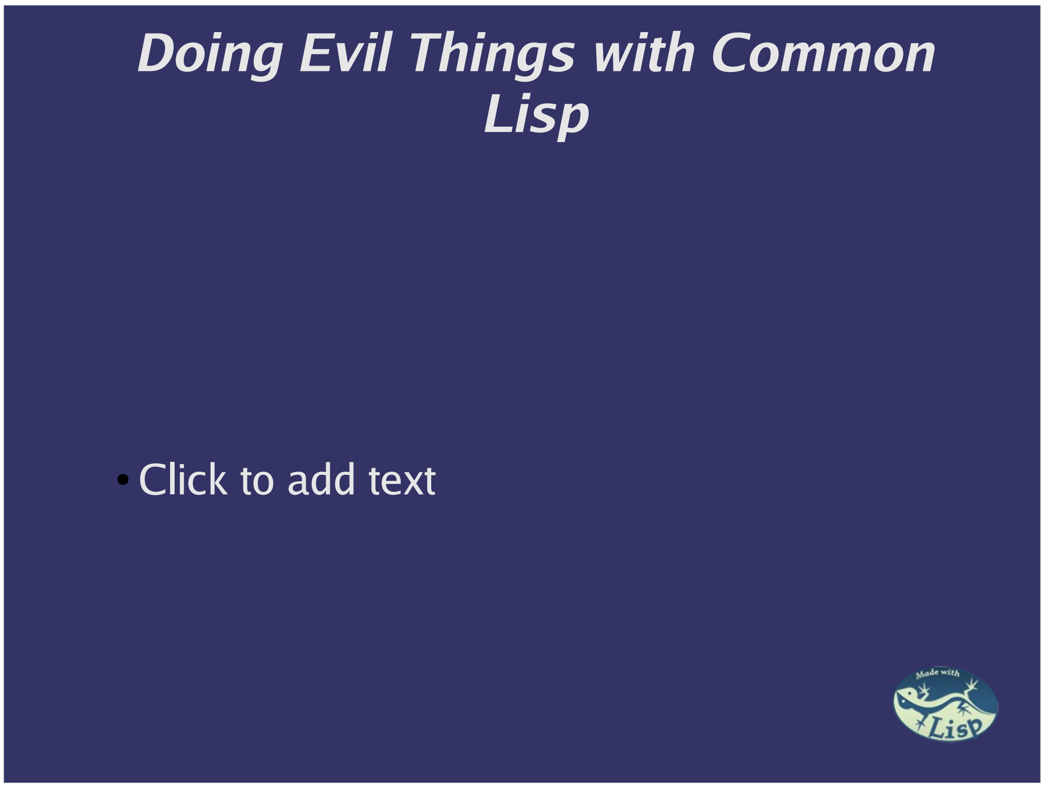
➲
```
(DEFCFUN intfunc INT
     (ARG1 INT))
```

Oh, that evil thing?

- Something I was playing with when Bill contacted me
  1. Create a CFFI C function to a name that doesn't exist, (defcfun foo :int (a :int))
  2. Create a callback function that matches that signature, (defcallback foo-lisp :int ((a :int)))
  3. Hack SBCL's foreign function linkage table to point foo at foo-lisp
  4. Call (FOO 4)
- Execution will do a convoluted jump from Lisp code, to a C calling convention, to Lisp code and back.
- Probably not useful :)

END

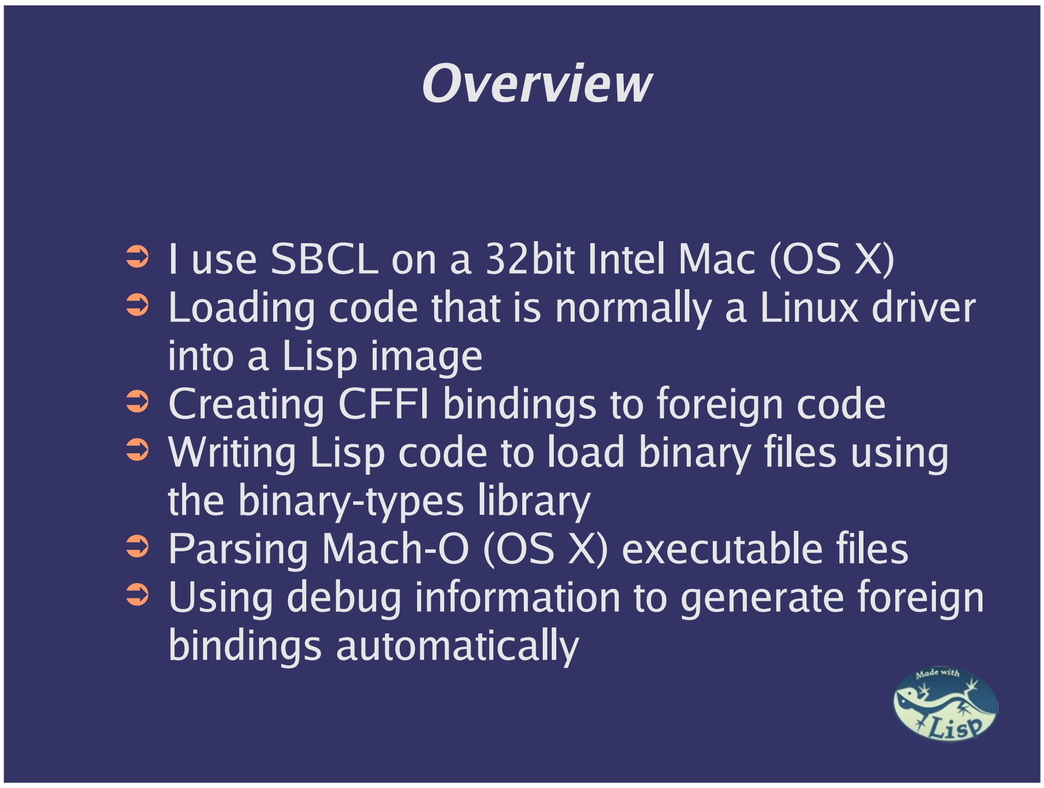
- Discussion (hopefully)



Doing Evil Things with Common Lisp

- Click to add text

A somewhat misleading title really, I promise that I'll explain it at the end. This presentation isn't really about a single piece of software or anything. Bill managed to convince me to present *something* and I was doing some interesting stuff. So this talk is about that interesting stuff.



Overview

- I use SBCL on a 32bit Intel Mac (OS X)
- Loading code that is normally a Linux driver into a Lisp image
- Creating CFFI bindings to foreign code
- Writing Lisp code to load binary files using the binary-types library
- Parsing Mach-O (OS X) executable files
- Using debug information to generate foreign bindings automatically

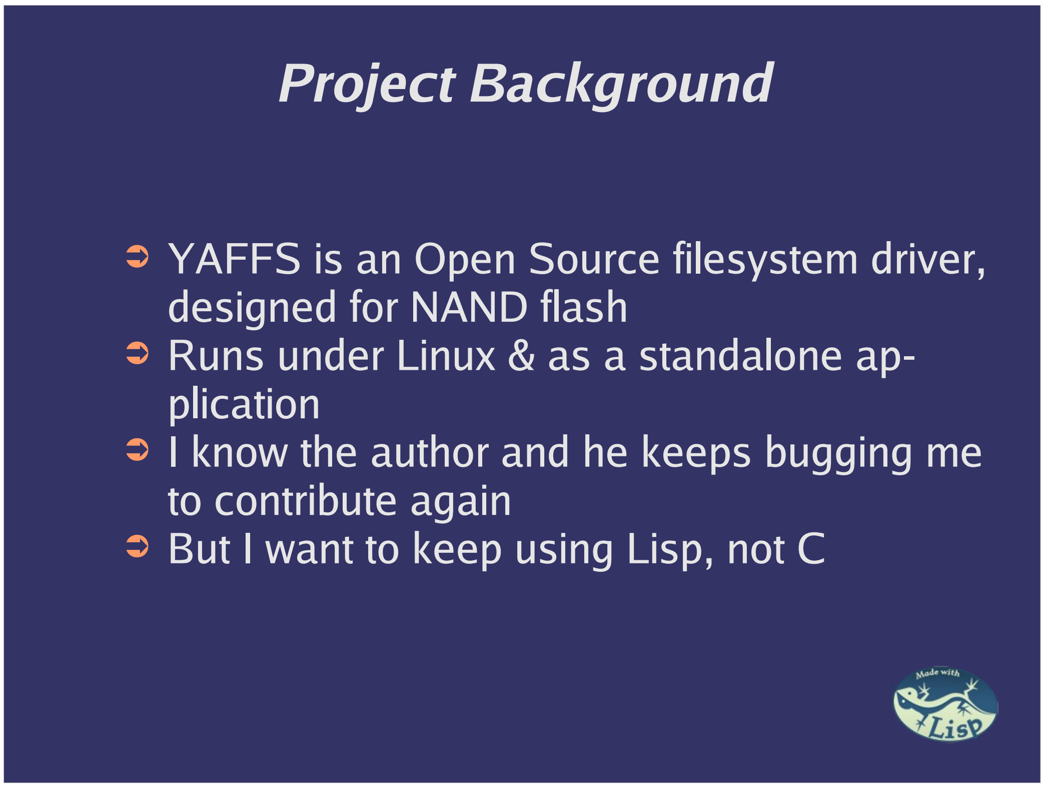
Made with
Lisp

My SBCL is a few months old now. The last point is IMHO the most interesting.

My Background

- 7 years of C/C++ experience doing embedded systems work
- 1 year of C/C++ in the games industry obsessing over performance and memory usage
- 2.5 years of Common Lisp with nothing of note to show for it & I still feel like a novice. But I'm having fun. Learn a language in 10 years and all that.

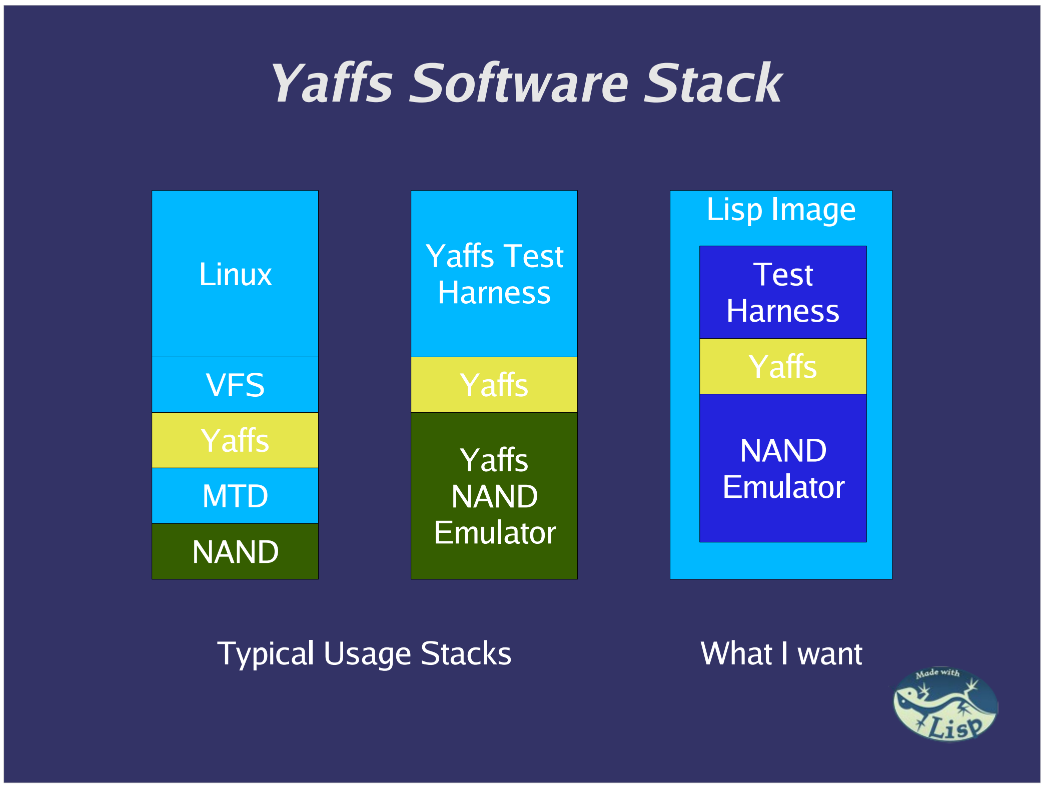
The embedded systems work was mostly WinCE and Linux. I wrote device drivers, tools, etc.



Project Background

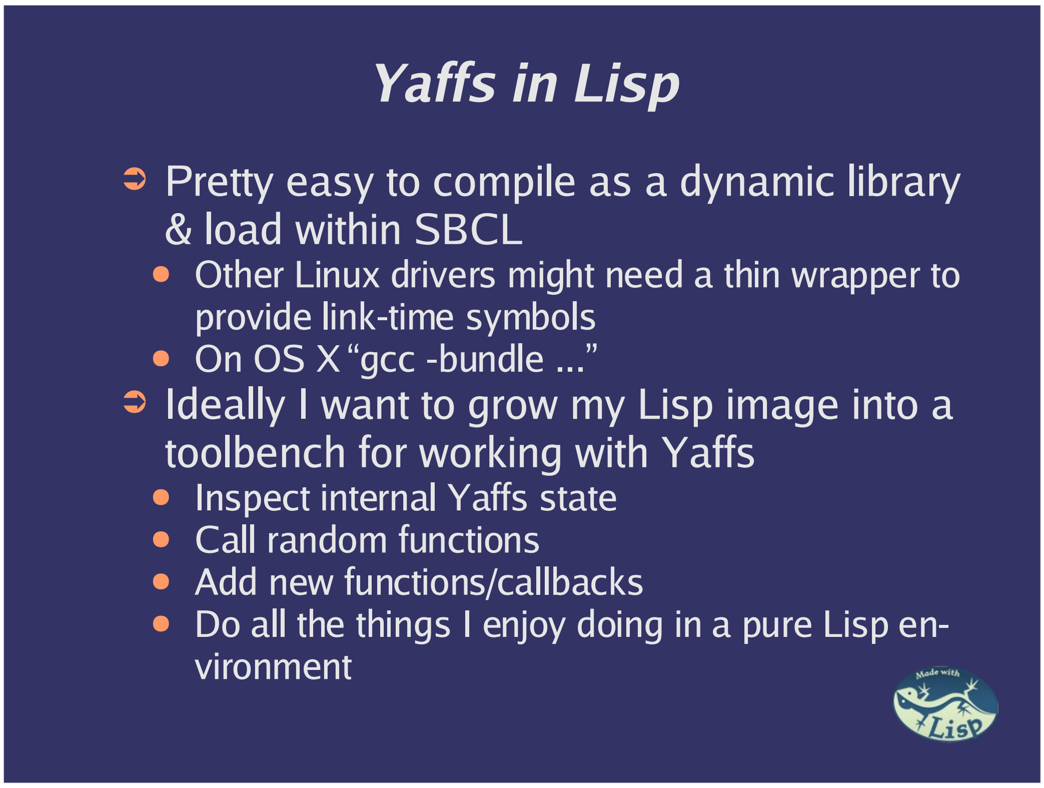
- YAFFS is an Open Source filesystem driver, designed for NAND flash
- Runs under Linux & as a standalone application
- I know the author and he keeps bugging me to contribute again
- But I want to keep using Lisp, not C

NAND flash is the most common non-volatile solid state storage. It's in iPods, camera cards, usb drives, etc. Apple buys more than 50% of all of it. NAND is different enough from other systems that a custom file system is pretty much required.



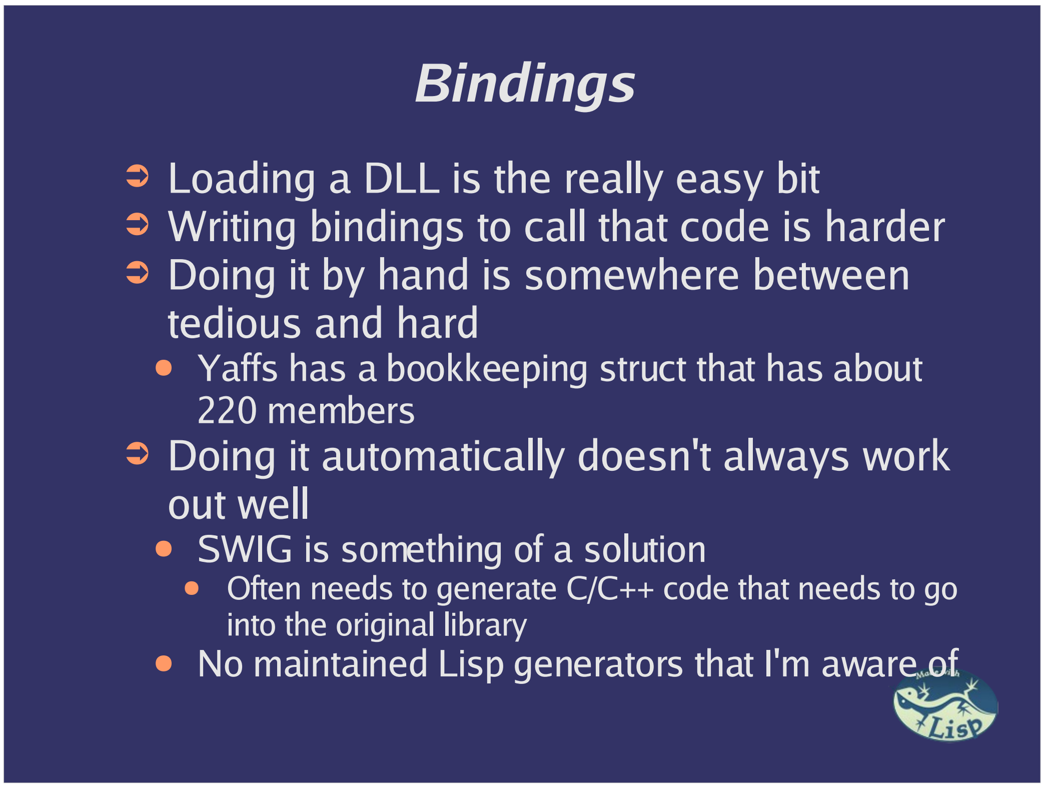
Yaffs Software Stack

Yaffs actually make it pretty easy to build as a stand alone library.
Yaffs provides a bunch of different low-level filesystem primitives, which any OS could hook into.
To work with Linux it also interfaces with VFS.



Yaffs in Lisp

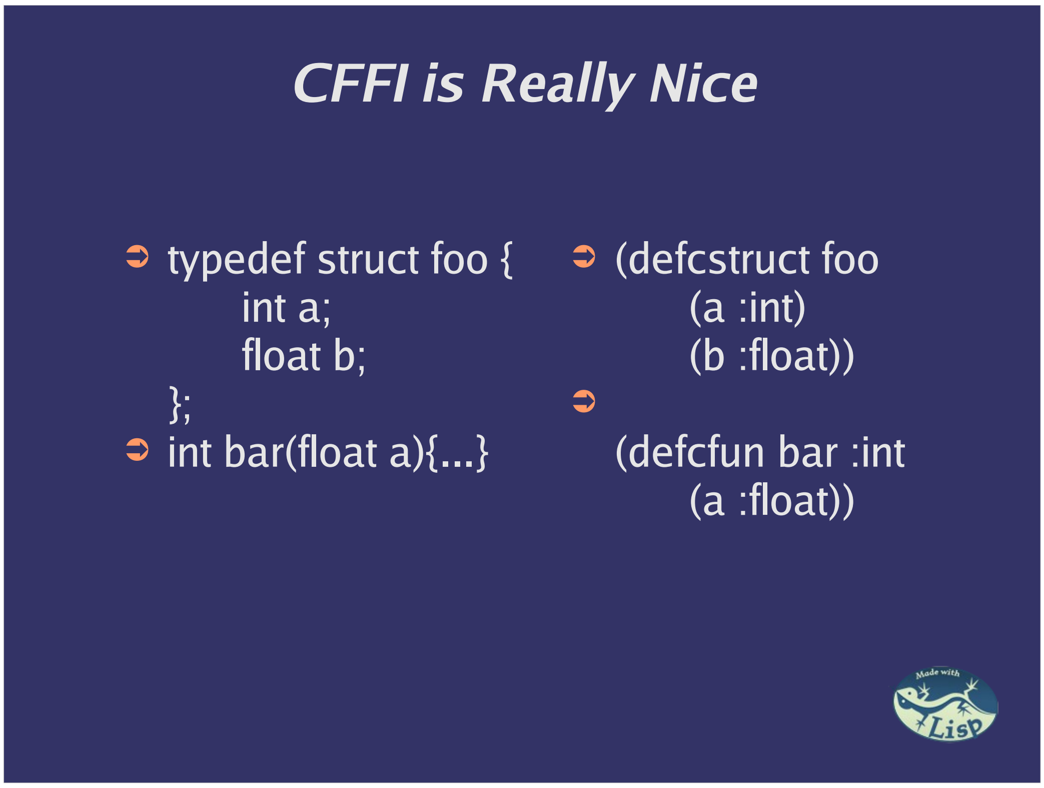
- Pretty easy to compile as a dynamic library & load within SBCL
  - Other Linux drivers might need a thin wrapper to provide link-time symbols
  - On OS X "gcc -bundle ..."
- Ideally I want to grow my Lisp image into a toolbench for working with Yaffs
  - Inspect internal Yaffs state
  - Call random functions
  - Add new functions/callbacks
  - Do all the things I enjoy doing in a pure Lisp environment



Bindings

- Loading a DLL is the really easy bit
- Writing bindings to call that code is harder
- Doing it by hand is somewhere between tedious and hard
  - Yaffs has a bookkeeping struct that has about 220 members
- Doing it automatically doesn't always work out well
  - SWIG is something of a solution
    - Often needs to generate C/C++ code that needs to go into the original library
  - No maintained Lisp generators that I'm aware of

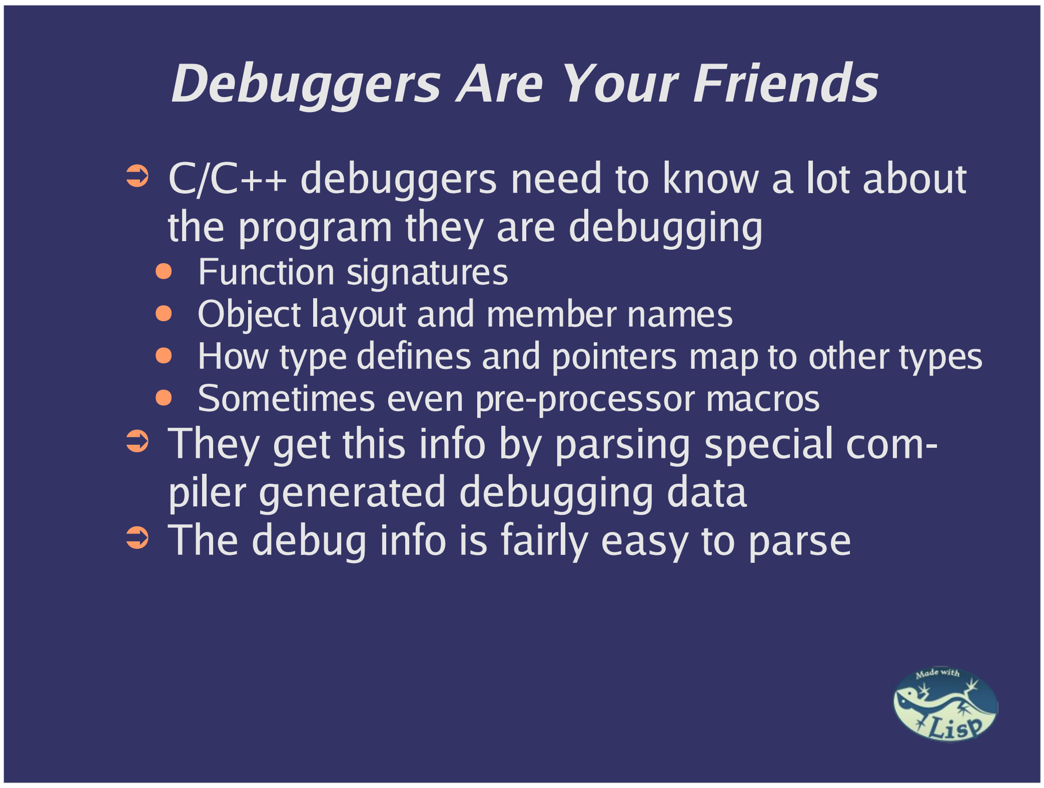
220 members!!!
SWIG is actually pretty good. But it needs to be folded into the make system, needs to get #defines right, has to parse multiple files, may need to be massaged to parse hairy C/C++ constructs. It needs to be linked against the original code so is somewhat intrusive.
The main problem with SWIG is that C++ is impossible to parse, existing compilers just fake it :)



CFFI is Really Nice

```
➲ typedef struct foo {        ➲ (defcstruct foo
        int a;                        (a :int)
        float b;                      (b :float))
  };                          ➲
➲ int bar(float a){...}         (defcfun bar :int
                                      (a :float))
```

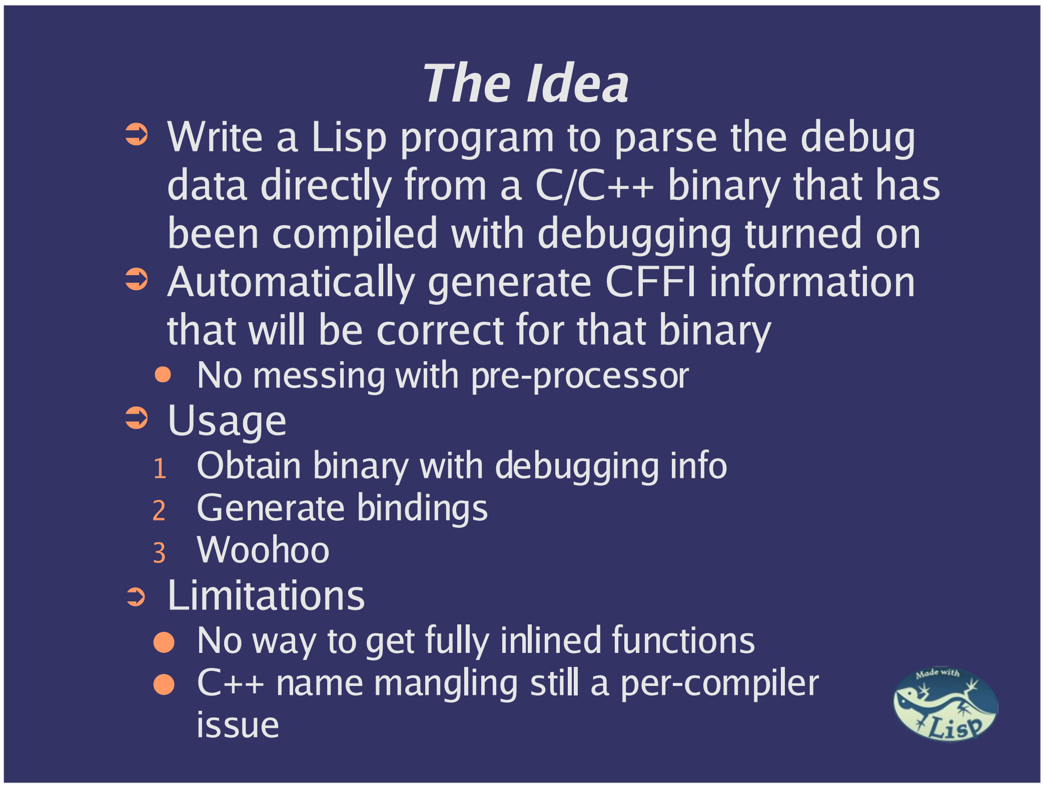
CFFI looks pretty much like a “minimal” representation of the C code, I don't see how you could encode the required information in a smaller way. That make CFFI really nice in my book.



Debuggers Are Your Friends

- C/C++ debuggers need to know a lot about the program they are debugging
  - Function signatures
  - Object layout and member names
  - How type defines and pointers map to other types
  - Sometimes even pre-processor macros
- They get this info by parsing special compiler generated debugging data
- The debug info is fairly easy to parse

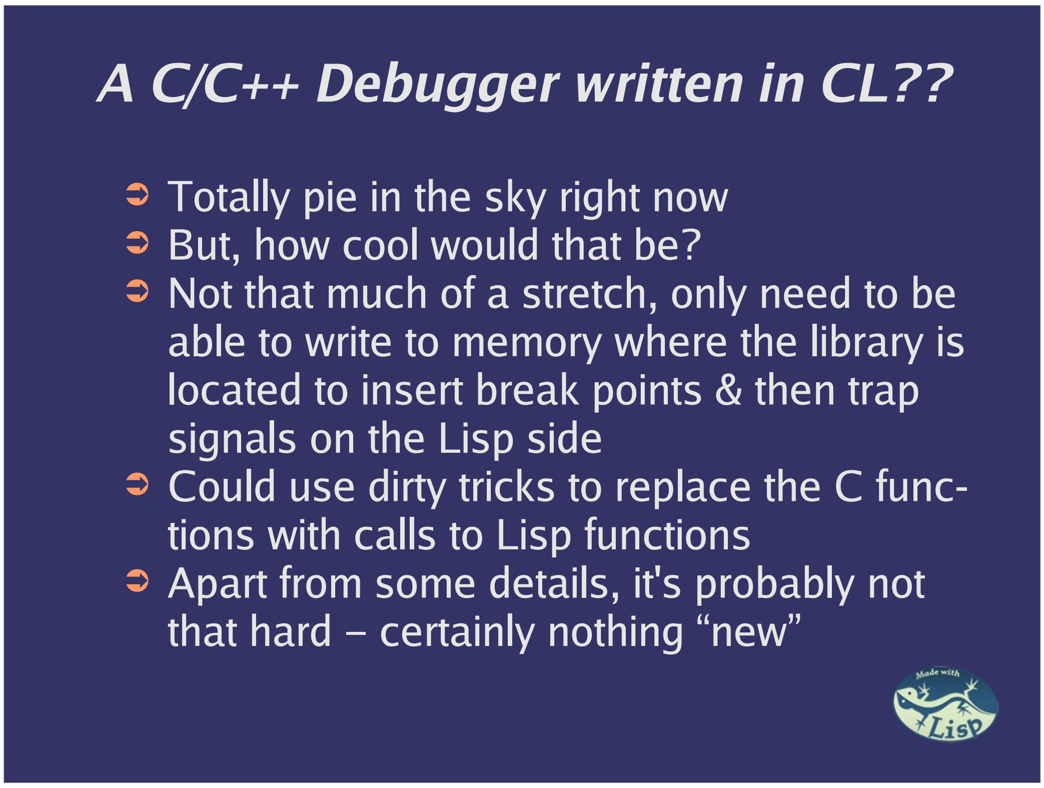
I've Googled and can't really find anybody else using this technique to generate scripting bindings. The best I found was a usenet posting from around 1996 that doesn't appear to have lead anywhere.
I'm not at all sure about macros ending up in the debug output, but I think I've read that somewhere.



The Idea

- Write a Lisp program to parse the debug data directly from a C/C++ binary that has been compiled with debugging turned on
- Automatically generate CFFI information that will be correct for that binary
  - No messing with pre-processor
- Usage
  1. Obtain binary with debugging info
  2. Generate bindings
  3. Woohoo
- Limitations
  - No way to get fully inlined functions
  - C++ name mangling still a per-compiler issue

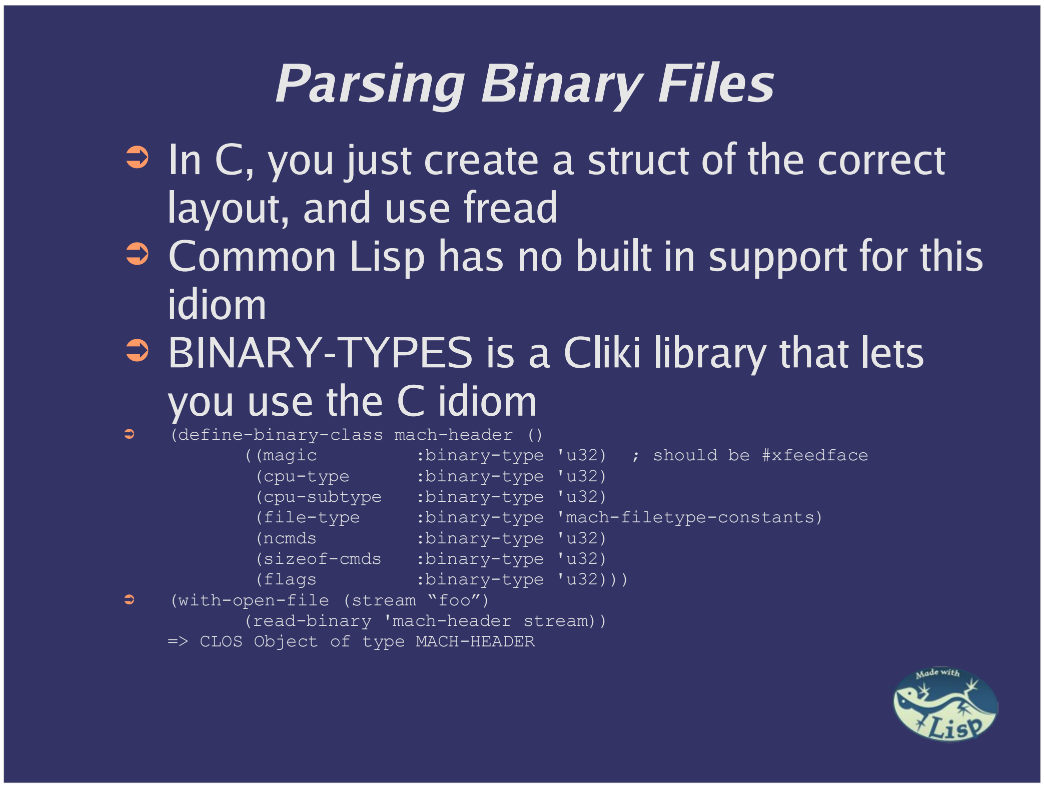
For large projects, the build system itself can be a nightmare to work with. Having to integrate SWIG into that could be a lot of work. Having all the required information in one file can make life easier.
C++ name mangling is just annoying, but I don't think it's very hard to deal with.



A C/C++ Debugger written in CL??

- Totally pie in the sky right now
- But, how cool would that be?
- Not that much of a stretch, only need to be able to write to memory where the library is located to insert break points & then trap signals on the Lisp side
- Could use dirty tricks to replace the C functions with calls to Lisp functions
- Apart from some details, it's probably not that hard – certainly nothing “new”

The dirty trick would involve overwriting the C code with another jump (branch without link) into a Lisp callback. When the Lisp code returns it can jump directly back to the original calling function via the link register.
Also, I'm totally glossing over the real difficulty in writing a debugger – it would be time consuming.



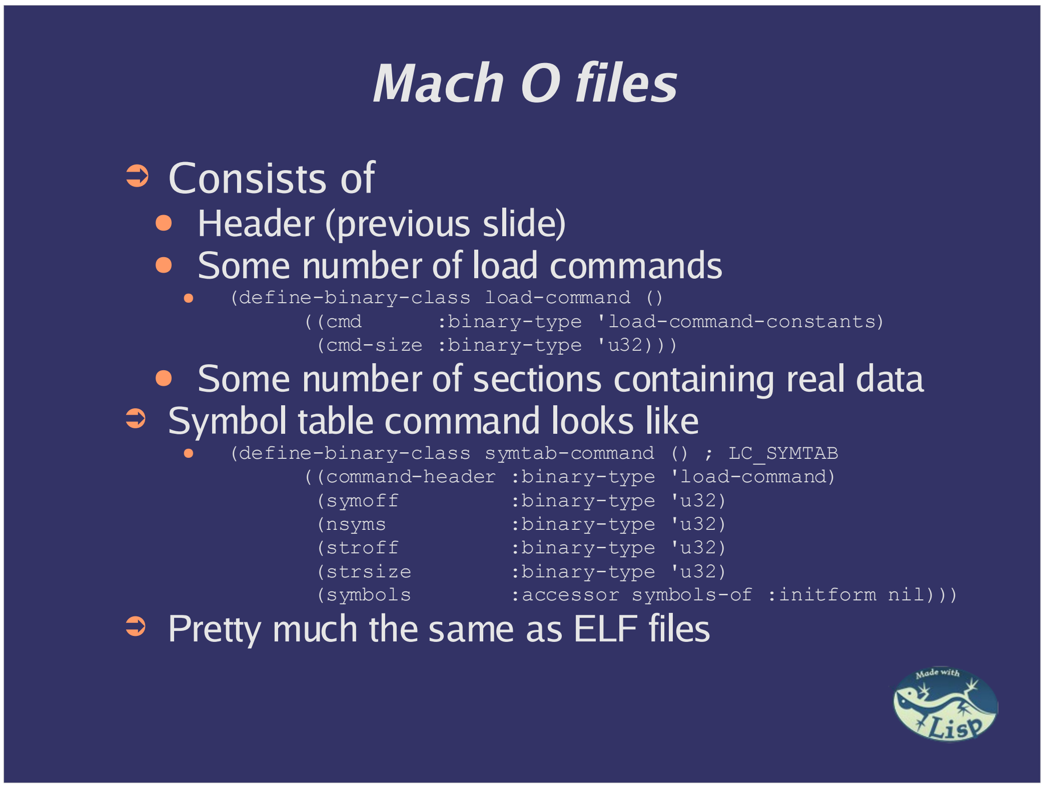
Parsing Binary Files

- In C, you just create a struct of the correct layout, and use fread
- Common Lisp has no built in support for this idiom
- BINARY-TYPES is a Cliki library that lets you use the C idiom

```
(define-binary-class mach-header ()
       ((magic         :binary-type 'u32)  ; should be #xfeedface
        (cpu-type      :binary-type 'u32)
        (cpu-subtype   :binary-type 'u32)
        (file-type     :binary-type 'mach-filetype-constants)
        (ncmds         :binary-type 'u32)
        (sizeof-cmds   :binary-type 'u32)
        (flags         :binary-type 'u32)))
```

```
(with-open-file (stream "foo")
       (read-binary 'mach-header stream))
=> CLOS Object of type MACH-HEADER
```

I'm aware of PCL's chapter on binary file reading, but it doesn't come in an easy to use package.
Also, I may be missing something obvious & CL may have a better idiom for binary file reading.

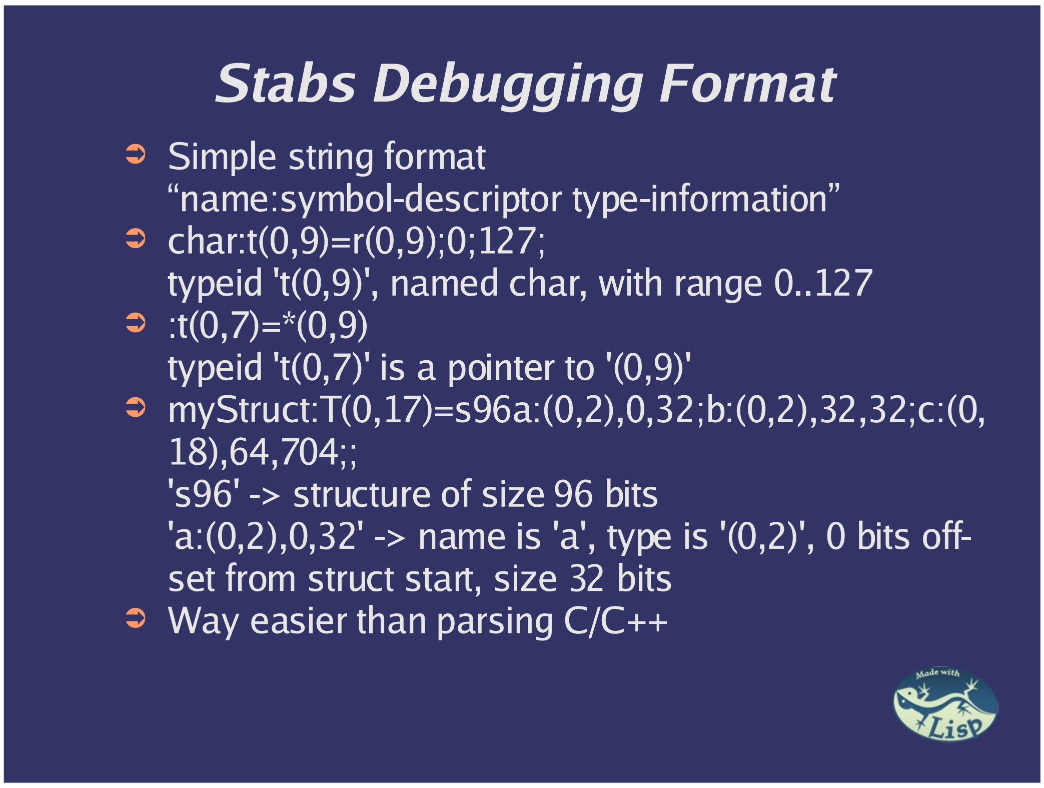


Mach O files

- Consists of
  - Header (previous slide)
  - Some number of load commands

    -
      ```
      (define-binary-class load-command ()
            ((cmd      :binary-type 'load-command-constants)
             (cmd-size :binary-type 'u32)))
      ```
  - Some number of sections containing real data
- Symbol table command looks like

  -
    ```
    (define-binary-class symtab-command () ; LC_SYMTAB
          ((command-header :binary-type 'load-command)
           (symoff         :binary-type 'u32)
           (nsyms          :binary-type 'u32)
           (stroff         :binary-type 'u32)
           (strsize        :binary-type 'u32)
           (symbols        :accessor symbols-of :initform nil)))
    ```
- Pretty much the same as ELF files



Stabs Debugging Format

- Simple string format
  “name:symbol-descriptor type-information”
- char:t(0,9)=r(0,9);0;127;
  typeid 't(0,9)', named char, with range 0..127
- :t(0,7)=*(0,9)
  typeid 't(0,7)' is a pointer to '(0,9)'
- myStruct:T(0,17)=s96a:(0,2),0,32;b:(0,2),32,32;c:(0,18),64,704;;
  's96' -> structure of size 96 bits
  'a:(0,2),0,32' -> name is 'a', type is '(0,2)', 0 bits off-set from struct start, size 32 bits
- Way easier than parsing C/C++

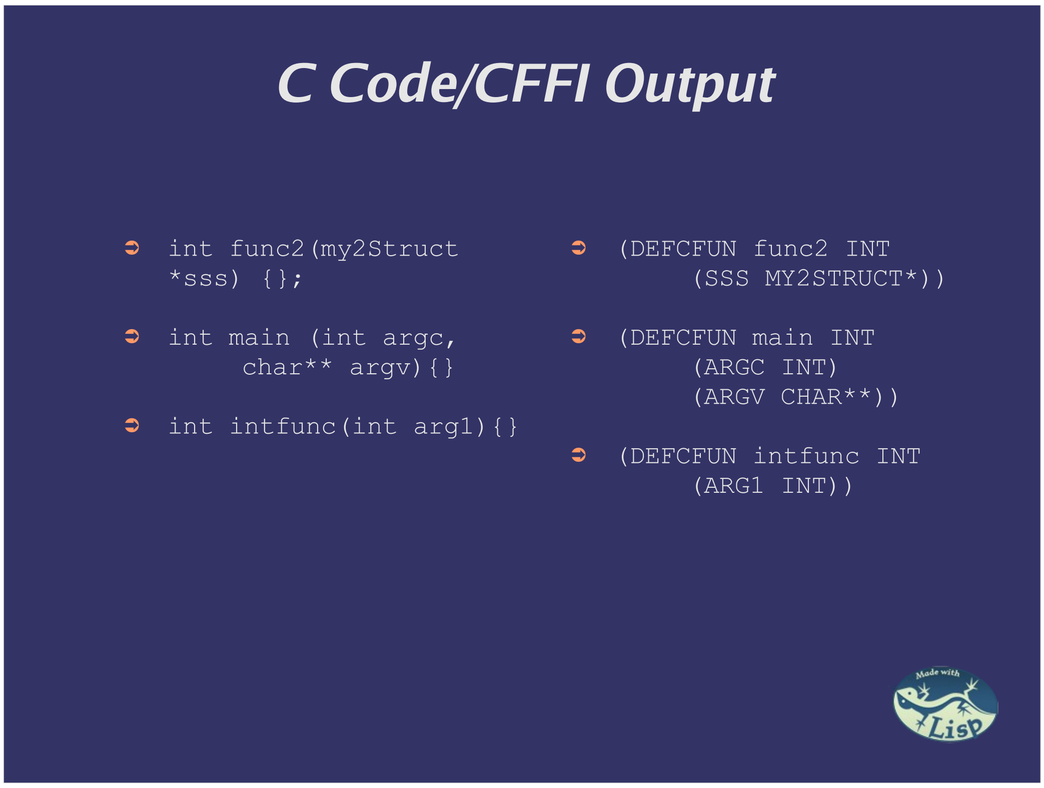
Made with Lisp

I've got a fairly complete C parser (it can parse the Yaffs binary) in < 500 SLOC. I'm re-writing it for clarity and size now.
When I'm happy with the C generation I'll start working on C++ generation.

C Code/CFFI Output

```
typedef int foo;
typedef char* string;
struct myStruct
{
    int a;
    int b;
    int c[22];
};

typedef struct
{
    int c;
} my2Struct;

typedef enum
{
    Efoo,
    Ebar
} foobar;

enum {
    foo2
} foobar2;

typedef union
{
    int i;
    char c;
} myunion;
```

```
(DEFCTYPE FOO INT)
(DEFCTYPE STRING CHAR*)
(DEFCSTRUCT MYSTRUCT
       (A INT OFFSET 0 COUNT 1)
       (B INT OFFSET 4 COUNT 1)
       (C INT OFFSET 8 COUNT 22))

(DEFCTYPE INT :int32)
(DEFCTYPE CHAR** :pointer)

(DEFCSTRUCT MY2STRUCT
       (C INT OFFSET 0 COUNT 1))


(DEFCENUM FOOBAR
       (EFOO 0)
       (EBAR 1))
(DEFCENUM ENUM-2003
       (FOO2 0))

(DEFCTYPE CHAR* :string)
(DEFCTYPE CHAR :char)
(DEFCTYPE FLOAT :float)
(DEFCTYPE DOUBLE :float)
(DEFCTYPE UNSIGNED-CHAR :uint8)

(DEFCUNION MYUNION
       (I INT COUNT 1)
       (C CHAR COUNT 1))
```



C Code/CFFI Output

-
  ```
  int func2(my2Struct
  *sss) {};
  ```

-
  ```
  int main (int argc,
       char** argv){}
  ```

-
  ```
  int intfunc(int arg1){}
  ```

-
  ```
  (DEFCFUN func2 INT
       (SSS MY2STRUCT*))
  ```

-
  ```
  (DEFCFUN main INT
       (ARGC INT)
       (ARGV CHAR**))
  ```

-
  ```
  (DEFCFUN intfunc INT
       (ARG1 INT))
  ```

Oh, that evil thing?

- Something I was playing with when Bill contacted me
  1. Create a CFFI C function to a name that doesn't exist, (defcfun foo :int (a :int))
  2. Create a callback function that matches that signature, (defcallback foo-lisp :int ((a :int)))
  3. Hack SBCL's foreign function linkage table to point foo at foo-lisp
  4. Call (FOO 4)
- Execution will do a convoluted jump from Lisp code, to a C calling convention, to Lisp code and back.
- Probably not useful :)

END

- Discussion (hopefully)