Graydon's MKC Talks Collection

All 58 talks from the MKC series

Talk 1 - MKC Series

"One-Day Compilers"

or

How I learned to stop worrying
and love static metaprogramming

graydon@redhat.com

Talk 2 - MKC Series

Greetings

This is a talk about rapidly constructing compilers for domain-specific languages (DSLs). It is a little dense. To get through it, I am going to start with a number of facts which I will simply state and assume as true.

Domain-specific languages are good
"nearly ideal" software system abstractions
better than commands, APIs, object frameworks
very hard to identify and refine
even harder to implement well

DSL implementations are typically
a lot of effort
poor quality
slow
unsafe
incorrect

Talk 3 - MKC Series

Evidence

Considering the number of DSLs which have been developed,
the number which have succeeded widely is disappointing.

Make
Lex & Yacc
TeX & Postscript
Octave
RPM
Magicpoint
CGEN & Sid

uh.. LD scripts?

It is especially bad if you consider
their minimal feature sets, and
how much effort they took to make.

Talk 4 - MKC Series

Goals

I am going to show you how to make a DSL compiler in a single sitting. It will be capable of:

consuming the DSL's concrete syntax
lexing and parsing
(you should already know how to do this)

constructing a semantic model
with all the bells and whistles
variables, functions, etc.
types and type inference
and strong safety properties
static compile-time checks
error messages with input coordinates

emitting native code
which might actually be fast

Talk 5 - MKC Series

Example

I am going to make a compiler for a small subset of the Make(1) language. It accepts simple Makefiles like this:

```
# my awesome example

pooka=fiddle faddle
zug=bleh $(pooka)

main: foo $(zug)
gcc -o $@ $<

foo: bar.o baz.o bleh.o $(zug)
gcc -o foo bar.o baz.o $(pooka) bleh.o
```

Talk 6 - MKC Series

Tools

I am going to use some weird tools.
They are not all from the FSF.
There is nothing magic about them.

Objective Caml (ocaml)
ML dialect
Functional / Imperative / OO language
From INRIA
LGPL runtime / QPL compiler

Camlp4
Ocaml pre-processor-pretty-printer
"like (defmacro ...) on steroids"

Cquot
C code quasiquoting for camlp4
written by me

Talk 7 - MKC Series

How it's going to work

Obviously we cannot write an entire compiler in one sitting
But we can steal parts of other compilers.

The two compilers we've got to work with are ocamlc and gcc.
We need to decide which to use, or whether to use both.

The decision depends on which runtime we'd prefer to use.
Ocaml runtime is
large
complex
relatively obscure
C runtime is
tiny
simple
universally available

I expect most people in this audience would prefer the C runtime.

Talk 8 - MKC Series

Getting to the C runtime

The Ocaml compiler and runtime is not for everyone.

Ocaml is really good at producing C code though.

So we can make a translator from Makefiles to Ocaml programs
which generate C code, then run those programs to get our C code.

The end-user, in fact, doesn't even need to know about ocaml.

You can just give them the C code.

This is what we're going to do with Makefiles.

Talk 9 - MKC Series

The Big Picture (textual)

We are going to stitch together a "virtual compiler"

Makefile is translated to Ocaml program
Lexed and Parsed
Ocaml AST produced

Ocaml program is fed into ocamlc
Typechecked and compiled

Compiled Ocaml program is executed
Produces a C program

C program is fed into gcc
Resulting object file depends on nearly nothing

Talk 10 - MKC Series

The Big Picture (graphical)

Isn't that incredibly stupid?
No, it's the Unix Philosophy in action
(Remember, we are on a budget here)

Talk 11 - MKC Series

Prelude
An Ocaml Phrasebook

Talk 12 - MKC Series

Ocaml Phrasebook (part 1)

Basic stuff

int int
string string
int [] int array
list 'a list
hash_map 'a 'b Hashtbl.t
pair ('a * 'b)

typedef struct { type point = {
int x; x: int;
int y; y: int;
} point; }

if (foo) { if foo then
bar (); bar ()
} else { else
baz (); baz ()
}

/* comment */ (* comment *)

As you can see, it's not too far from home.

Talk 13 - MKC Series

Ocaml Phrasebook (part 2)

Functions

```
int foo (int bar) { let foo bar = bar + 1
return bar + 1;
}
```

Wait! Where are all the int type terms?
They are inferred them from the use of + 1

```
template let foo bar = bleh bar
T foo (T x) {
return bleh (x);
}
```

Yes, it infers polymorphic types too.

Talk 14 - MKC Series

Ocaml Phrasebook (part 3)

All the usual functional fun

```
(lambda (x) (+ x 1)) (fun x -> x + 1)
```

```
(lambda (x) (fun x ->
(lambda (y) (x y))) (fun y -> x y))
```

```
(foldl '+ 1 '(1 2 3)) List.foldl (+) 1 [1; 2; 3]
```

And all the usual procedural fun

```
for (i=0; i < 10; i++) { for i = 0 to 10 do
bleh (i); bleh i
} done
```

```
try { try
bleh (); bleh ()
} catch (exn x) { with
fixit (x); Exn x -> fixit x
}
```

Talk 15 - MKC Series

Ocaml Phrasebook (part 4)

Type terms are a little different

typedef foo bar; type bar = foo

typedef enum { peas, type veg = Peas
kale, | Kale
corn } veg; | Corn

Note: constructor names start with Upper Case Letters,
and type names start with lower case letters.

typedef list foo; type foo = int list

template type 'a foo = { member: 'a }
class foo {
T & member;
}

// invalid polymorphic C++ (* valid polymorphic ocaml *)
template type 'a bar = 'a foo
typedef foo bar ;

Talk 16 - MKC Series

Ocaml Phrasebook (part 5)

Unions are different, and it's important.

```
template type 'a tree =
union tree { Branch of ('a tree * 'a tree)
pair < tree &, | Node of 'a
tree & > branch;
T & node;
};
```

Why is the difference important?
Because the union elements on the right are disjoint.
You cannot treat a Node x as a Branch (x, y).
You have to match a 'a tree value against its constructors.
Like this:

```
match somevalue with
Branch (x,y) -> frobnicate_branch x y
| Node x -> frobnicate_node x
```

Talk 17 - MKC Series

Ocaml Phrasebook (part 6)

Ocaml is mostly assignment-free. This means that
a let creates a permanent (immutable) binding.

```
int x = 10; let x = 10 in
x = 11; /* assignment */ x = 11 (* boolean; false *)
```

If you want to use assignment, you need to explicitly
ask for a "reference" variable, which can have its value
accessed with ! or changed with :=

```
int x = 10; let x = ref 10 in
foo (x) foo (!x);
x = 11; x := 11
```

It is a little awkward, but it encourages you to isolate,
and understand, the state you use in a function. You tend
to need assignment a lot less than you'd assume.

Note: there are no "null" references. You cannot segfault.

Talk 18 - MKC Series

Ocaml Phrasebook (final bits)

The ; character is a separator, not terminator.

```
foo (); foo ();
bar (); bar ();
baz (); baz ()
```

Some idioms are worth translating

```
foo (zug *z) { let foo z = match z with
if (z) { Some x -> (* "ok" *)
/* non-NULL means OK */ | None -> (* "nothing" *)
} else {
/* NULL means "nothing" */
}
```

```
List *ls = hd; let foo ls = match ls with
while (l) { [] -> () (* ignore end *)
frob x; | x::xs -> (frob x; foo xs)
l = l->nxt;
}
(* or just... *)
let foo = List.iter frob
```

Talk 19 - MKC Series

Act 1
Consumption

Talk 20 - MKC Series

Front End (part 1)

Some easy stuff to warm up.
Makefiles have a pretty simple lexical structure.
Our lexer talks to camlp4 so we need to return ("CLASS","lexeme") pairs.
This is the entire lexer. It is written in ocamllex language.

```
let special = ['$' '@' '<' '^' '(' ')' '=' ':' '%']
let nq = ("\\\""|[^""])
let sym = ['-' '_' '.' '/' 'a'-'z''A'-'Z''0'-'9']*

rule token = parse
' '* { token lexbuf }
| ('#' [^'\n']*)? ('\n'+) { ("EOL", "") }
| '\t' { ("TAB", "") }
| eof { ("EOF", "") }
| (sym | "" nq* "") { ("WORD", lexeme lexbuf) }
| special { ("", lexeme lexbuf)}
| _ { ("", lexeme lexbuf)}
```

Talk 21 - MKC Series

Front End (part 2)

More easy stuff.
Makefiles have a pretty simple syntactic structure.
Our parser is recursive-descent, extended LL(1).
It is written in the pa_extend extension language of camlp4
This language is designed for extending the Ocaml grammar.
But we are just going to replace the grammar altogether.

```
EXTEND
makefile_item:
[ [ EOL -> None
| s = WORD; "="; ws = words; EOL -> ...
| t = WORD; ":"; ws = words; EOL; a = actions -> ... ] ];
actions: [ [ az = LIST0 action -> ... ] ];
action: [ [ TAB; ws = words; EOL -> ... ] ];
words: [ [ ws = LIST0 word -> ... ] ];
word: [ [ w = WORD -> ...
| "$"; "^" -> ...
| "$"; "<" -> ...
| "$"; "@" -> ...
| "$"; "("; w = WORD; ")" -> ... ] ];
END
```

Talk 22 - MKC Series

Front end (part 3)

Let's ignore the ... sections on the previous slide for now.
All we've got so far is a lexer and parser.
The parser builds "something" that represents the Makefile.

Now we have to decide
What to build
How to build it
What to do with the thing we build

Talk 23 - MKC Series

Semantic Model (part 1)

Our next step will be to build a semantic model
of the contents of a Makefile.

A collection of variables
Each var, a list of strings
doggies = dingo.o poodle.o
Or variable references
animals = llama.o $(doggies) loris.o
Referenced varialbes flatten
= llama.o dingo.o poodle.o loris.o

A dependency tree
Each node has a target and some dependencies
thezoo: $(animals) $(food) tourism.h
And an action to remake it
gcc -o $@ $(animals) $(food)
An action is a list of shell commands
which are just nested string lists, as with variables

Talk 24 - MKC Series

Semantic Model (part 2)

Assume for the moment that we have a technique for synthesizing any Ocaml program we like, given a Makefile as input.

We will get to that technique later.

For now, we're going to focus on building a semantic model of a Makefile by embedding it in an Ocaml program.

By performing this embedding, we will get a lot of semantic analysis from the Ocaml compiler for free
Variable binding
Rich native datatypes (lists, strings, etc.)
Functions
Lexical environments
Type inference and type checking

Talk 25 - MKC Series

Semantic Model (part 3)

We will embed the variable model directly in Ocaml semantics
Makefile string
Ocaml list with single string
Makefile variable
Ocaml function returning flattened list
Makefile variable reference
Ocaml function call

There is an obscure type-inference reason why functions are used here instead of variables.
When in doubt, try it with functions.

Example Makefile Fragment:
doggies = dingo.o poodle.o
animals = llama.o $(doggies) loris.o
...

Becomes Ocaml Functions:
```
let rec
doggies _ = List.flatten [["dingo.o"]; ["poodle.o"]]
and animals _ = List.flatten [["llama.o"]; doggies (); ["loris.o"]]
...
```

Talk 26 - MKC Series

Semantic Model (part 4)

For the dependency/action tree, we will define
an auxiliary Ocaml recursive datatype:

```
type file = string
type actions = string list
type rule = Rule of (file * rule list * actions)
```

Example Makefile Fragment:

```
lisp.o: lisp.c
gcc -c $<
emacs: emacs.c lisp.o buffer.h
gcc -o $@ emacs.c lisp.o
```

Becomes Ocaml Value:

```
Rule ("emacs",
[ Rule ("emacs.c", [], []);
Rule ("lisp.o",
[ Rule ("lisp.c", [], []) ],
[ "gcc -c lisp.c" ] );
Rule ("buffer.h", [] []); ],
[ "gcc -o emacs emacs.c lisp.o" ] )
```

Talk 27 - MKC Series

Semantic Model (part 5)

Ok, that was a bit of a simplification.

What we're really going to do is construct a function which constructs the rule tree. The example is more like this:

```
let rec
lisp_o _ =
let targ = "lisp.o" in
let dep_names = List.flatten [ ["lisp.c"] ] in
let deps = List.map resolve dep_names in
let actions = [String.concat " "
(List.flatten [ ["gcc"]; ["-c"];
[List.hd dep_names] ]) ] in
Rule (targ, deps, actions)
and emacs _ =
let targ = "emacs" in
let dep_names = List.flatten [ "emacs.c"; "lisp.o"; "buffer.h"] in
let deps = List.map resolve dep_names in
let actions = [String.concat " "
(List.flatten [ ["gcc"]; ["-o"]; targ;
["emacs.c"]; ["lisp.o"] ]) ] in
Rule (targ, deps, actions)
```

Talk 28 - MKC Series

Semantic Model (summary)

Make sure you have a semantic model worked out
Capture the meaning of your input language
Keep the model simple!
Translation is tricky without one
You won't get much typechecking either

Embed in native datatypes when possible
Lists, Strings, Tuples, Numbers, etc.
Variables
Functions

Make new types and constructors when necessary (or helpful)
Ocaml datatypes are recursive, branching, polymorphic
Very language-like
All user-visible errors will use your type names
Rename types when useful (eg: "file")

Talk 29 - MKC Series

Act 2
Plagiarism

Talk 30 - MKC Series

Quoting (part 1)

So far, we have been talking about consuming code.
Lexing, parsing, modelling semantics.

Next, we will be dealing with producing code.
Computing Abstract Syntax Trees (ASTs).

Our front-end will compute Ocaml ASTs for ocamlc.
To take advantage of ocamlc's semantic analysis.

Our back-end will compute C ASTs for gcc.
To take advantage of gcc's code generator and runtime.

ASTs are very tedious to write down.
In fact, even small fragments of ASTs are tedious to write down.

Talk 31 - MKC Series

Quoting (part 2)

For example, the AST for this Ocaml code:

List.map (fun x -> x+1) [1; 2; 3]

Is written (in Ocaml) as this value:

```
ExApp (loc, ExApp (loc, ExAcc (loc,
ExUid (loc, "List"), ExLid (loc, "map")),
ExFun (loc, [(PaLid (loc, "x"), None,
ExApp (loc, ExApp (loc, ExLid (loc, "+"),
ExLid (loc, "x")), ExInt (loc, "1")))])),
ExApp (loc, ExApp (loc, ExUid (loc, "::"), ExInt (loc, "1")),
ExApp (loc, ExApp (loc, ExUid (loc, "::"),
ExInt (loc, "2")), ExApp (loc,
ExApp (loc, ExUid (loc, "::"), ExInt (loc, "3")),
ExUid (loc, "[]")))))
```

Which is completely tedious to write down.

Talk 32 - MKC Series

Quoting (part 3)

We are are trying to avoid tedious stuff.

Luckily, two groups of people have seen this problem before.
Lisp hackers
Think about(defmacro ...)
Logicians
Think about proving programs

They both invented the same thing: quoting.

Quoting (v):

1. Denoting an AST by providing a piece
of text from the language you are trying
to construct.

Talk 33 - MKC Series

Quoting (part 4)

The way quoting works is simple and dirty.

You want an AST for some code:

List.map (fun x -> x+1) [1; 2; 3]

So you write that code in magic quotes:

<:expr< List.map (fun x -> x+1) [1; 2; 3] >>

Then you run it through a pre-processor.
The pre-processor substitutes the AST of the quoted code.
The compiler never sees the magic quotes.
It is as though you wrote the AST there instead.

Talk 34 - MKC Series

Quoting (part 5)

In lisp
The magic quotes are (quote ...)
also written as '...
The pre-processor is built in to the lisp reader
You can only quote lisp
which if fine, for macros

In Ocaml
The magic quotes are <:foo< ... >>
where foo is an "expander name"
The pre-processor is camlp4
You can quote any language with an expander
You can add expanders

Talk 35 - MKC Series

Quoting (part 6)

The quoted language is called the Object Language.
(the word "object" has nothing to do with OOP here)

The quoting language is called the Meta Language.
(subtle suggestion: ponder what the ML stands for in Ocaml)

Talk 36 - MKC Series

Quoting (final exam)

When we quote Ocaml code in Ocaml, like this:

```
let my_fn = <:expr< List.iter (fun x -> x+1) [1; 2; 3] >>
```

It is NOT equivalent to writing this:

```
let my_fn = List.iter (fun x -> x+1) [1; 2; 3]
```

But rather, it is equivalent to writing this:

```
let my_fn = ExApp (loc, ExApp (loc, ExAcc (loc,
ExUid (loc, "List"), ExLid (loc, "iter")),
ExFun (loc, [(PaLid (loc, "x"), None,
ExApp (loc, ExApp (loc, ExLid (loc, "+"),
ExLid (loc, "x")), ExInt (loc, "1")))])),
ExApp (loc, ExApp (loc, ExUid (loc, "::"), ExInt (loc, "1")),
ExApp (loc, ExApp (loc, ExUid (loc, "::"),
ExInt (loc, "2")), ExApp (loc,
ExApp (loc, ExUid (loc, "::"), ExInt (loc, "3")),
ExUid (loc, "[]")))))
```

Talk 37 - MKC Series

Quoting (summary, Q&A;)

Quoting makes synthesizing ASTs completely trivial.
Avoid it at your own peril.

Q: What if my Object language has the lexeme >> in it?
A: Escape it: \>\>

Q: What if there is a parse error in a quotation?
A: The pre-processor will emit an error.

Q: Doesn't pre-processing distort source co-ordinates?
A: No, the pre-processor preserves source co-ordinates.

Q: Is my code parsed a second time after pre-processing?
A: No, the pre-processor and compiler are tightly integrated.

Talk 38 - MKC Series

Quasiquoting (part 1)

Quoting is OK for entering ASTs you know already.
But you don't always know all ASTs in advance.
If you did, you wouldn't need a compiler.

At best, you know parameterized ASTs.
AST templates, in other words, with gaps in them.

Terminology:

A normal quoted AST is called a quotation.
A quotation with gaps to fill in is called a quasiquotation.
A gap in a quasiquotation is called an antiquotation.

Talk 39 - MKC Series

Quasiquoting (part 3)

In camlp4, an antiquotation is written as $ty:var$

It means "put the AST named var, of type ty, here".

For example, writing:

```
let ast1 = <:expr< (fun y -> y + 1) >>
let ast2 = <:expr< List.map $expr:ast1$ [1; 2; 3] >>
```

Is equivalent to writing:

```
let ast2 = <:expr< List.map (fun y -> y + 1) [1; 2; 3] >>
```

Or in lisp, using backquote:

```
(define ast1 '(lambda (y) (+ y 1)))
(define ast2 `(map ,ast1 (1 2 3)))
```

Talk 40 - MKC Series

Quasiquoting (final exam)

Here is a more complex example, with C ASTs.
This is an Ocaml function which builds ASTs that represent
"calling a function on every value of an N-entry array"

```
let iterate n arr fn =
<:cstmt<
for (i = 0; i < $int:n$; i++) { $ident:fn$($expr:arr$[i]); }
>>
```

The pre-processor expands this to:

```
let iterate n arr fn =
FOR
(BINARY
(ASSIGN, VARIABLE "i", CONSTANT (CONST_INT "0")),
BINARY
(LT, VARIABLE "i",
CONSTANT (CONST_INT (string_of_int n))),
UNARY (POSINCR, VARIABLE "i"),
COMPUTATION
(CALL (VARIABLE fn, [INDEX (arr, VARIABLE "i")])))
```

Talk 41 - MKC Series

Quasiquotations (summary)

Quasiquotations permit combination of ASTs, without having to give up the convenience of quotation.

With camlp4 we can make new types of quasi and anti quotations.

Whenever you are faced with having to build ASTs, start by making sure you have a good collection of quotations for your object language.

In the remaining section, we are going to complete the compiler.
The front end will produce quotations of Ocaml code.
The back end will produce quotations of C code.

Emacs hates quotations.

Talk 42 - MKC Series

Act 3
Regurgitation

Talk 43 - MKC Series

Front end (part 4)

Recall, we left a number of unfinished ... things in our parser.

```
word:
[ [ w = WORD -> ...
| "$"; "^" -> ...
| "$"; "<" -> ...
| "$"; "@" -> ...
| "$"; "("; w = WORD; ")" -> ... ]
];
```

We now fill these in with Ocaml quasiquotations.
They come from our Ocaml model of Makefile semantics.
Aren't you glad we worked that out in advance?

```
word:
[ [ w = WORD -> <:expr< [$str:w$] >>
| "$"; "^" -> <:expr< dep_names >>
| "$"; "<" -> <:expr< [List.hd dep_names] >>
| "$"; "@" -> <:expr< [targ] >>
| "$"; "("; w = WORD; ")" -> let w2 = tidy w in
<:expr< ($lid:w2$ ()) >> ]
];
```

Talk 44 - MKC Series

Front end (part 5)

Now we fill in the quasiquotations for the makefile_item parser:

```
makefile_item:
[ [ EOL -> None
| s = WORD; "="; ws = words; EOL ->
Some (s, <:expr< $ws$ >>, ASSIGN)
| t = WORD; ":"; ws = words; EOL; a = actions ->
let ex = <:expr<
let targ = $str:t$ in
let dep_names = $ws$ in
let deps = List.map resolve dep_names in
let actions = $a$ in
Mk.Rule (targ, deps, actions) >>
in Some (t, ex, RULE) ] ];

actions: [ [ az = LIST0 action -> (mk_list az loc) ] ];
action: [ [ TAB; ws = words; EOL ->
<:expr< String.concat " " $ws$ >> ] ];
words: [ [ ws = LIST0 word ->
(let ls = mk_list ws loc in
<:expr< List.flatten $ls$ >>) ] ];
```

Talk 45 - MKC Series

Front end (part 6)

Now we have a parser which constructs an AST for an Ocaml function corresponding to an item in a Makefile (an assignment or a rule).

We are going to replace the core Ocaml grammar with a grammar that reads a sequence of Makefile items, builds Ocaml functions, and emits an AST containing the sequence of functions and a single call into our backend with the result of those functions.

We are then going to plug this grammar into camlp4 and ocamlc, and we'll be done the front end.

The AST node Pcaml.implem is the root of the Ocaml grammar.
It represents a single compilation unit in Ocaml terminology.
It is what we need to make, to feed to ocamlc.

Talk 46 - MKC Series

Front end (finale)

Here is the code to build the Pcaml.implem AST node.
This part is very boring. It is about 20 more lines, and is very technical.
You aren't expected to read this now, just see that it is rather short.

```
Pcaml.implem: [ [ res = makefile -> ([(res, loc)], False) ] ];

makefile: [ [ maybe_items = LIST0 makefile_item; EOF ->
let items = some maybe_items in
let rule_p (_, _, x) = match x with RULE -> true | ASSIGN -> false in
let (rules, assigns) = partition rule_p items in
let dead_rule = (<:patt< x >>, None, <:expr< Mk.Rule (x, [], []) >>) in
let live_rules = map (fun (n,_,_) -> let n1 = tidy n in
(<:patt< $str:n$ >> , None, <:expr< $lid:n1$ () >>)) rules
in
let resz = live_rules @ [dead_rule] in
let resolver = (<:patt< resolve >>, <:expr< fun [ $list:resz$ ] >>) in
let bind (n, e, _) = let pwel = [<:patt< _ >>, None, e] in
let n1 = tidy n in <:patt< $lid:n1$ >> , <:expr< fun [$list:pwel$] >>
in
let (r_binds, a_binds) = (map bind rules, map bind assigns) in
let funs = resolver :: (a_binds @ r_binds) in
let (top,_,_) = hd rules in
let tree = <:expr< let rec $list:funs$ in $lid:top$ () >> in
let entry = [( <:patt< _ >>, <:expr< Be.trans $tree$ >> )] in
let recur = false in
let items = [ <:str_item< value $rec:recur$ $list:entry$ >> ] in
<:str_item< declare $list:items$ end >> ] ];
```

Talk 47 - MKC Series

Back end (part 1)

Recall that the compiler we are building is a 2-stage system.

Front End
Lex and parse Makefile
Syntax checking happens here
Synthesize Ocaml program
Compile Ocaml program with ocamlc
Static semantic checking happens here

Back End
Run compiled Ocaml program
Capture synthesized C program
Compile C program with gcc

We have completed the guts of the front end.
The back end really only requires us to write one function.
It is the function that calculates a C program from an Ocaml value.

Talk 48 - MKC Series

Back end (part 2)

Recall also that our Ocaml program will produce a single value
of the following rule type:

```
type file = string
type actions = string list
type rule = Rule of (file * rule list * actions)
```

For example:

```
Rule ("emacs",
[ Rule ("emacs.c", [], []);
Rule ("lisp.o",
[ Rule ("lisp.c", [], []) ],
[ "gcc -c lisp.c" ] );
Rule ("buffer.h", [] []); ],
[ "gcc -o emacs emacs.c lisp.o" ] )
```

Which we must translate into a C program

Talk 49 - MKC Series

Back end (part 3)

The C program will do "the work" of a Makefile.
The recursive algorithm (vaguely) is:

```
int check_node_foo_c (time_t parent) {
time_t mtime = 0;
int rebuild = 0;
struct stat target;
if (stat ("foo.c", ■) == -1)
rebuild = 1;
else
mtime = statr.st_mtime;
rebuild = check_first_dep (mtime) || rebuild;
// ...
if (rebuild) {
write (1, "gcc -c foo.c\n", 13);
system ("gcc -c foo.c");
return 1
}
else
return mtime > parent;
}
```

With some extra error handling thrown in.

Talk 50 - MKC Series

Back end (part 4)

Now that we've worked out the structure of the C program we're going to generate, we just write it down as a bunch of factored quasiquotations. Here is an example quote from the back end:

```
let fstmt = <:cstmt<
if (stat ($str:file$, ■) == -1) {
rebuild = 1;
} else {
mtime = target.st_mtime;
}
$stmt:satisfy$
>>
```

Look familliar?

The back end is about 70 lines of code. It just produces C quotes isomorphic to the rule tree constructed by the front end.

Talk 51 - MKC Series

Back end (summary)

The back end is probably the easiest part to understand, since we've already developed a keen understanding of quasiquotation.

Produces a C function for each rule
Trades space for speed
Not that much space either: ~ 30 insns / rule at -O2
Pre-calculates a lot
Only needs stat(), system(), and write()
Many other tradeoffs possible

Can perform domain-specific optimizations
Topological sort done at compile time
Missing build rules directly fail when called
Dead build rules not compiled into executable
Common dependency DAG nodes refer to same function
All messages pre-formatted
All string lengths known, for write()

Talk 52 - MKC Series

The Driver (part 1)

All that remains is to write a driver:

Run ocamlc on input, with with custom camlp4 pre-processor
Chmod output file
Run output file, capture output to C file
Run gcc on C file
Chmod output file
Delete temporaries

Some command-line option processing and exception handling,
make the driver about as large as the back end.

Talk 53 - MKC Series

The Driver (part 2)

We use the "shell" module of Ocaml, since it looks a lot like writing shell scripts with flexible, argument processing, I/O and exception handling. It is pretty straightforward:

```
...
call [cmd "ocamlc" ["-pp"; homedir ^ "/mkc-fe -impl";
"-o"; runfile; "-I"; homedir;
"-I"; "frontc"; "frontc.cma"; "be.cmo";
"-impl"; makefile ]];
Unix.chmod runfile 0o700;
call ~stdout:(to_file cfile) [cmd runfile []];
call [cmd "gcc" ["-O2"; "-o"; outfile; cfile]];
Unix.chmod outfile 0o755;
...
```

Look familliar?

Talk 54 - MKC Series

Finale
&
Complaints

Talk 55 - MKC Series

Summary

We produced a compiler for a domain specific language.

By "compiler", we mean at least:

Lexing and Parsing
Variable binding
Type checking
Error reporting
Native code generation

And we did it in under 400 lines of code.

Talk 56 - MKC Series

Q&A; (part 1: shameless promotion)

Q: Do I have to learn some icky language to do this?
A: Yes. It's really not that icky.

Q: Aw, couldn't we just have done this in m4 or a Perl script?
A: Not without losing a lot of convenience and safety.

Q: Does Ocaml have:
a debugger, a profiler, lex and yacc?
arrays, for loops, printf, and fast I/O?
generics, modules, and typesafety?
hashtables, objects, and garbage collection?
map, fold, lambda, cons, and quote?
an emacs mode and a free, portable compiler?
A: Yup.

Q: Why am I still coding in C/C++/Java/Lisp?
A: Beats me.

Talk 57 - MKC Series

Q&A; (part 2: this particular compiler)

Q: Why a Makefile compiler? Isn't that dumb?
A: It's a simple language everyone knows.

Q: Can I use this for huge Makefiles with lots of funny functions?
A: No. You probably don't want to use it. It's still pretty minimal.

Q: So you lied! This isn't a "one-day compiler" at all, is it?
A: For a small DSL, a day (or a few) is enough to get it going.

Q: And for larger languages?
A: You may find your language semantics make the embedding harder.

Q: Where can I get the code to play with it?
A: http://www.venge.net/cgi-bin/cvsweb.cgi/venge/code/src/mkc/

Q: Is this part of Red Hat's secret plan?
A: No. I just do this on my own time.

Talk 58 - MKC Series

Fini