

# interactive code checking with **Cobra**

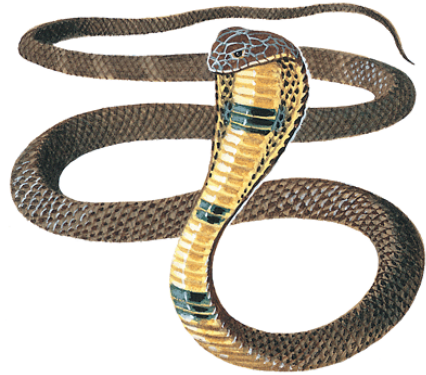
a Tutorial

Gerard Holzmann

Nimble Research

[gholzmann@acm.org](mailto:gholzmann@acm.org)

this course



*Code Browser  
and Analyzer*

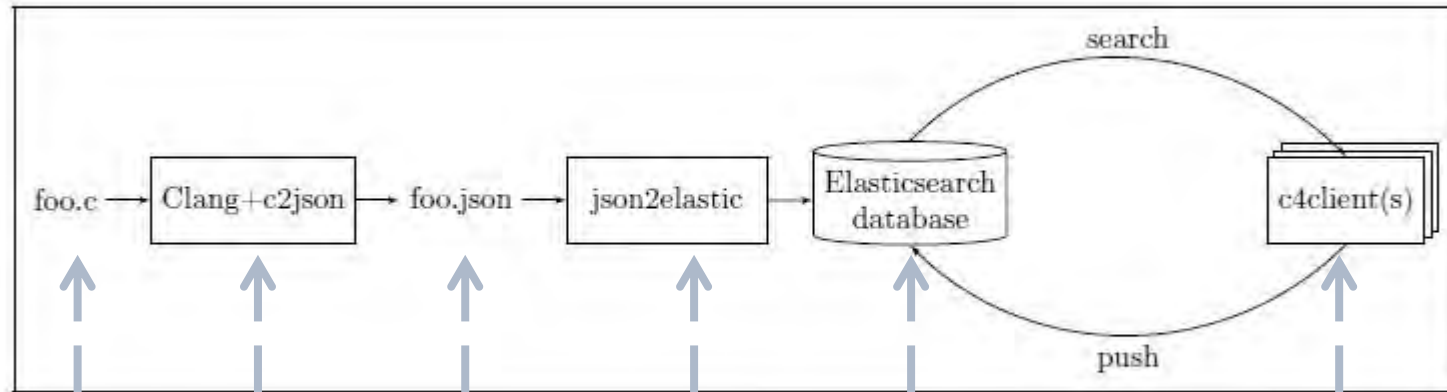
- why was it built?
- what can it do?
- how does it work?
- how can you use it?

# synopsis

- *context*
  - principle of operation
  - installation and configuration
  - query libraries
  - guide to online documentation
- *pattern search*
  - patterns and regular expressions
- *interactive queries*
  - query command language
  - token attributes, token maps
  - sets and ranges
  - defining functions, reading files
  - visualization, scaling
- *inline scripting*
  - basic scripting method
  - keywords, control-flow primitives
  - defining recursive functions
  - using associative arrays
  - concurrency
- *building standalone checkers*
  - the cobra api
  - use multi-threading

# background: a challenge project

designing & building an interactive code query system



**Figure 1:** Overall C4 architecture. `c2json` and `json2elastic` are responsible for generating the indexing ASTs into an Elasticsearch database. Users use the `c4client` API to ask questions about the codebase (`search`) and store conclusions (`push`).

*code*

*compile*

*AST*

*convert*

*web  
database*

*query  
processing*

# the challenge project

## summer internship project report

### 4 Results

We tested C4's capabilities by first writing a generic call graph analysis. This analysis pushes a 2 directional call graph into C4.

Second we wrote analyses that leveraged the result of the call graph to determine which public interface functions of Europa's Core FSW codebase and the MSL codebase sent which interprocess messages. Although we did not record times for Europa's Core FSW codebase, we did record times for certain modules in the MSL codebase (table 1). For these benchmarks, C4 used a single node (12 shard) Elasticsearch database running on a 12-core processor with 30 gigabytes of heap space.

Lines of Code	Functions	Call Graph Time	Interprocess Message Time
13321	297	8.16 min	18.28 sec
56429	812	1.28 hr	2.55 min
64843	1159	1.33 hr	1.92 min

**Table 1:** Call graph and interprocess message times for three MSL modules. As expected, analysis time increased as the number of lines of code / functions increased. Most work was done during the CallGraph analysis.

*much* too slow, especially when targeting millions of lines of code

# the challenge project

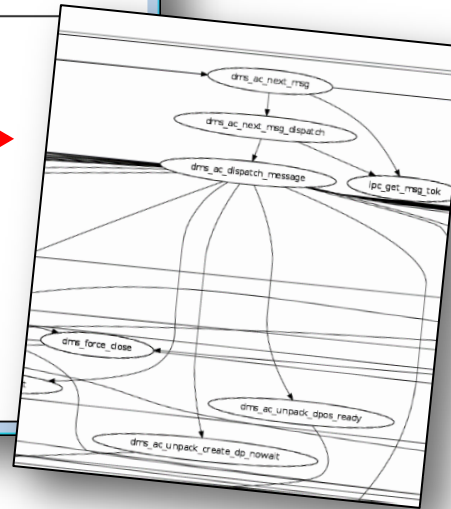
it doesn't have to be that slow: computing the function call graph

MSL Module	Lines of Code	Functions	Call Graph Time	Interprocess Message Time
files	13321	297	8.16 min	18.28 sec
cbm	56429	812	1.28 hr	2.55 min
dms	64843	1159	1.33 hr	1.92 min

**Table 1:** Call graph and interprocess message times for three MSL modules. As expected, analysis time increased as the number of lines of code / functions increased. Most work was done during the CallGraph analysis.

Cobra on the  
MSL dms  
source code

```
gh@nada: ~/msltest_gh/src/dms
Main Options VT Options VT Fonts
$ cat *.[ch] | ncs1
  sloc   ncs1 comments
 64035  43028  19540
$ time ( echo "fcg" | cobra -n *.[ch] )
260879
: wrote: cobra.dot (857 nodes, 4525 edges)
view with: !dot -Tx11 cobra.dot &
0 matches
:
: real    0m0.341s
: user    0m0.304s
: sys     0m0.032s
$
```



857 functions, 4525 function calls  
call graph generation: 0.3 seconds

# the challenge project

it doesn't have to be that slow: checking which MSL IPC message types are sent

MSL Module	Lines of Code	Functions	Call Graph Time	Interprocess Message Time
files	13321	297	8.16 min	18.28 sec
cbm	56429	812	1.28 hr	2.55 min
dms	64843	1159	1.33 hr	1.92 min

Cobra

these -

```
gh@nada: ~/msltest_gh/src
Main Options VT Options VT Fonts
$ cat */*.[ch] | ncs1
sloc ncs1 comments
158061 95293 32680
$ time ( echo ". %C/cobra_scripts/ipc2.cobra" | cobra -n */*.[ch] ) | awk '{ print $3 }' | sort -u

'DMS_AC_IPC_PRI_CLONE_REQUEST'
'DMS_AC_IPC_PRI_COMP'
'DMS_AC_IPC_PRI_FILES'
'DMS_AC_IPC_PRI_FSM'
'DMS_AC_IPC_PRI_OTHER'
'DMS_AC_IPC_PRI_PDP'
'DMS_AC_IPC_PRI_VDP'
'DMS_AC_IPC_PRI_XS'
'FILES_AC_IPC_PRI_ASYNC'
'FILES_AC_IPC_PRI_INTERNAL'
matches
'NVMCAMMGR_AC_IPC_PRI_CAM_CMD'
'NVMCAMMGR_AC_IPC_PRI_CAM_ENGINE'
'NVMCAMMGR_AC_IPC_PRI_CANCEL'
'NVMCAMMGR_AC_IPC_PRI_MONITOR'
'NVMCAMMGR_AC_IPC_PRI_NAND_CMD'
'NVMCAMMGR_AC_IPC_PRI_NOR_CMD'
'NVMCAMMGR_AC_IPC_PRI_NOTIFY'
'NVMCAMMGR_AC_IPC_PRI_REQUEST'
'NVMCAMMGR_AC_IPC_PRI_SDRAM_CMD'
'PDP_AC_IPC_PRI_REQUEST_REALTIME'
'PDP_AC_IPC_PRI_REQUEST_RECORDED'
'PDP_AC_IPC_PRI_TIMER'
'TIM_AC_IPC_PRI_TIM_CMD_PRIORITY'
'TIM_AC_IPC_PRI_TIM_DMS_CMD_PRIORITY'
'TIM_AC_IPC_PRI_TIM_LOAD_SC_TIME_CMD_PRIORITY'
'TIM_AC_IPC_PRI_TIM_NPI_SAVE_CMD_PRIORITY'
'TIM_AC_IPC_PRI_TIM_NPI_NOTIF_MSG_PRIORITY'

real 0m0.734s
user 0m0.676s
sys 0m0.060s
$
```

← includes construction of db

*cobra query commands sequence:*  
mark ipc\_send  
mark ipc\_check\_and\_send  
next,  
next  
list

# performance can be critical

consider this real scenario from early in the MSL mission

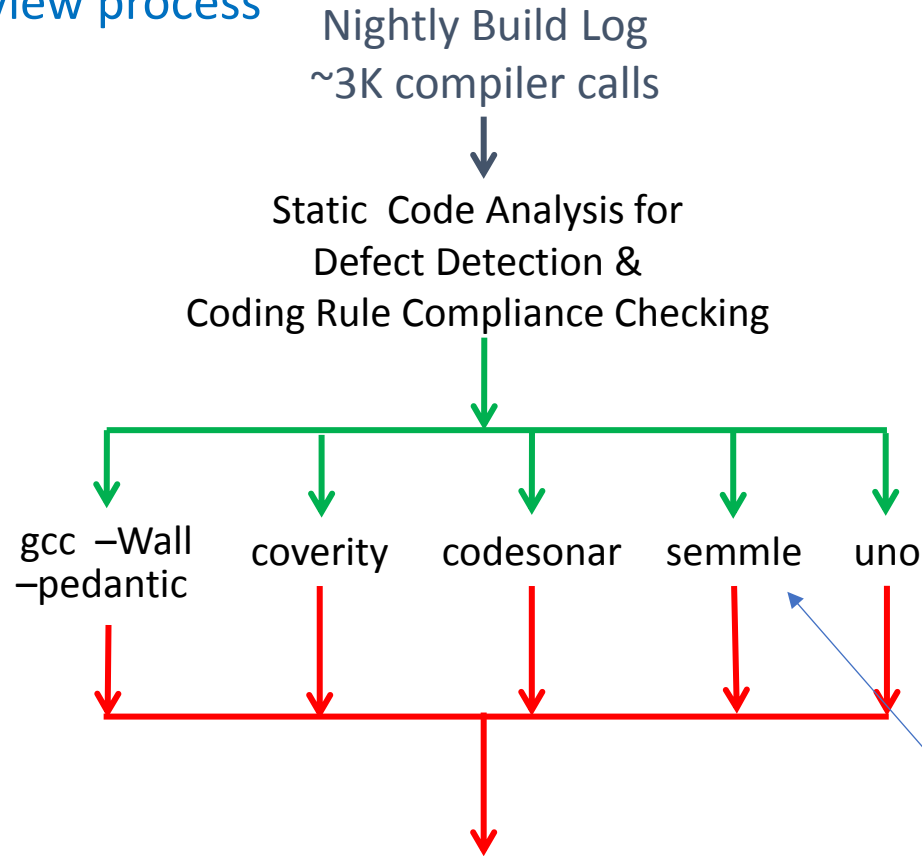


- an in-flight anomaly occurs
  - manual analysis reveals the cause:
  - a function call passes an array argument of the wrong size
    - function expects an array of 16 elements
    - the call passes an array of 8 elements
    - data corruption results (compilers don't catch this)
- *does this happen anywhere else in the 2.8 MLOC?*
  - old method:
    - develop a new checker for (one of) the static analyzers
    - wait 15 hours for the cumulative check to be run
    - meanwhile, a few million miles away.....



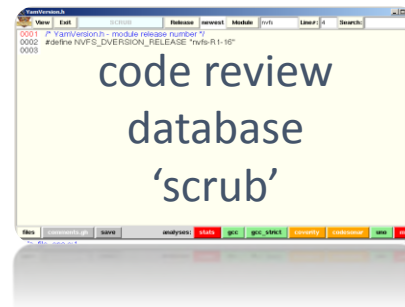
# performance can be critical

JPL tool-based code review process



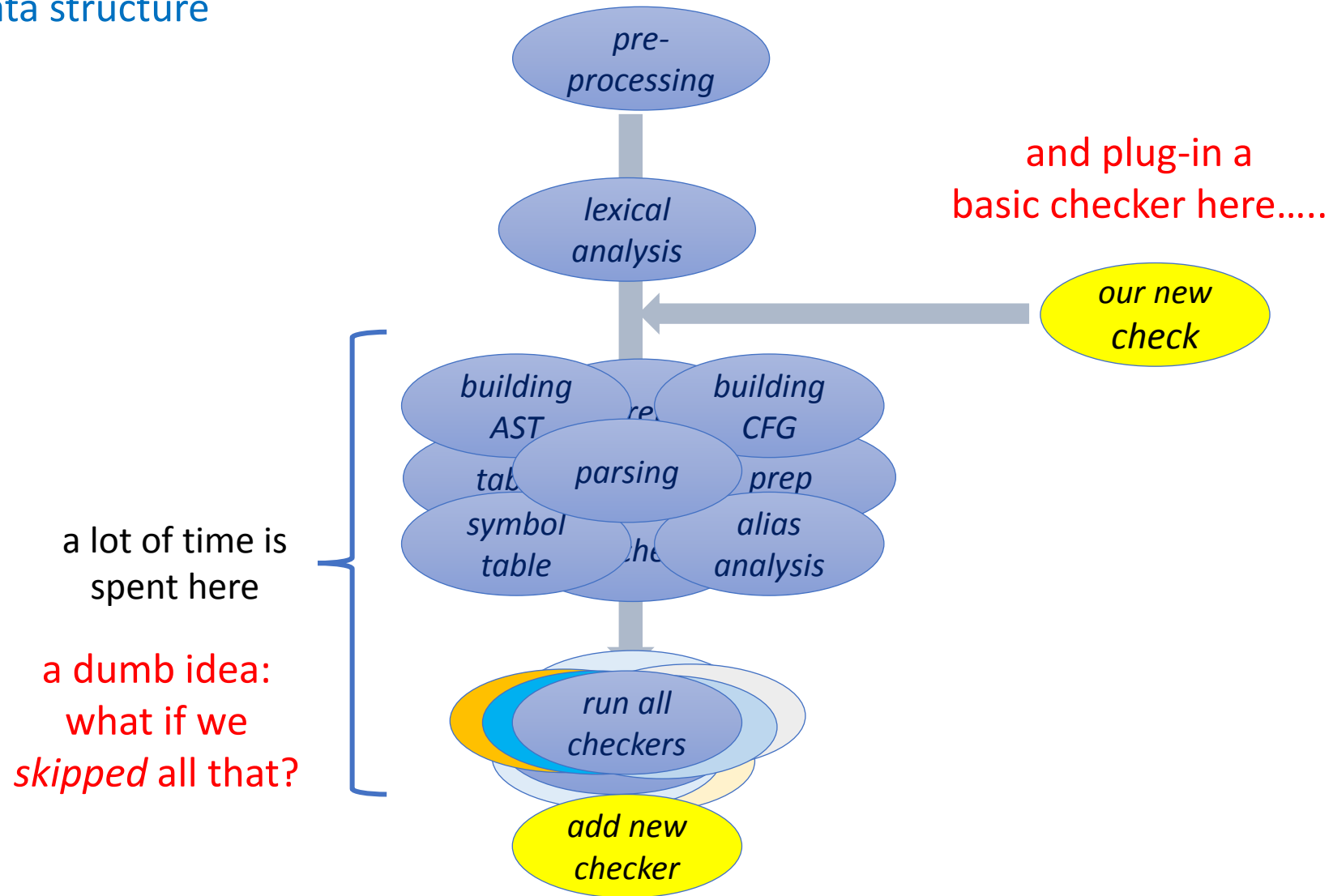
analysis time for  
2.8 MLOC of C  
~15hrs

*in this case, we asked semmle  
to build a new checker for us,  
which they delivered the next day  
so that we could add it to the nightly check*



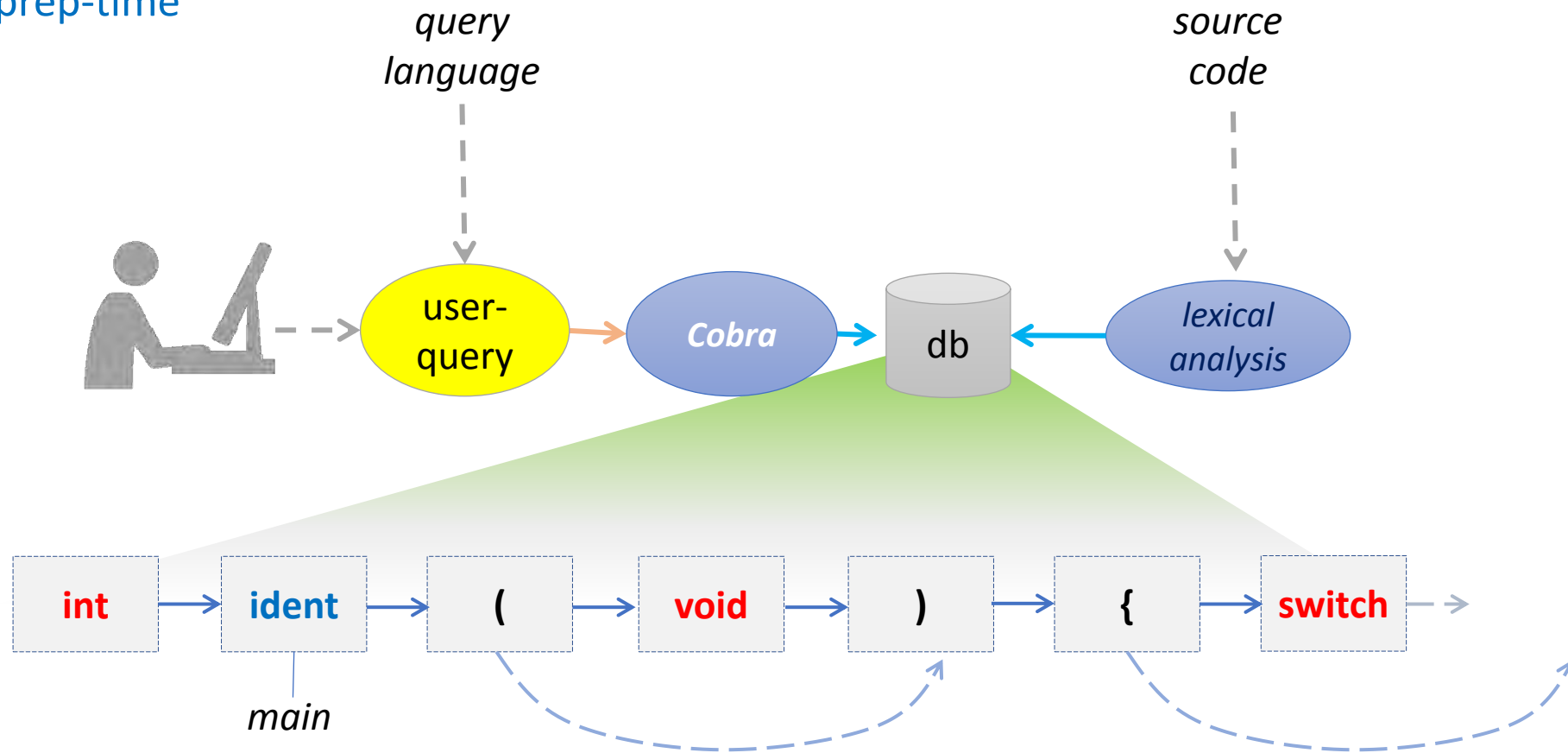
# why does the analysis take so long?

building data structure



# cobra's design

minimize prep-time

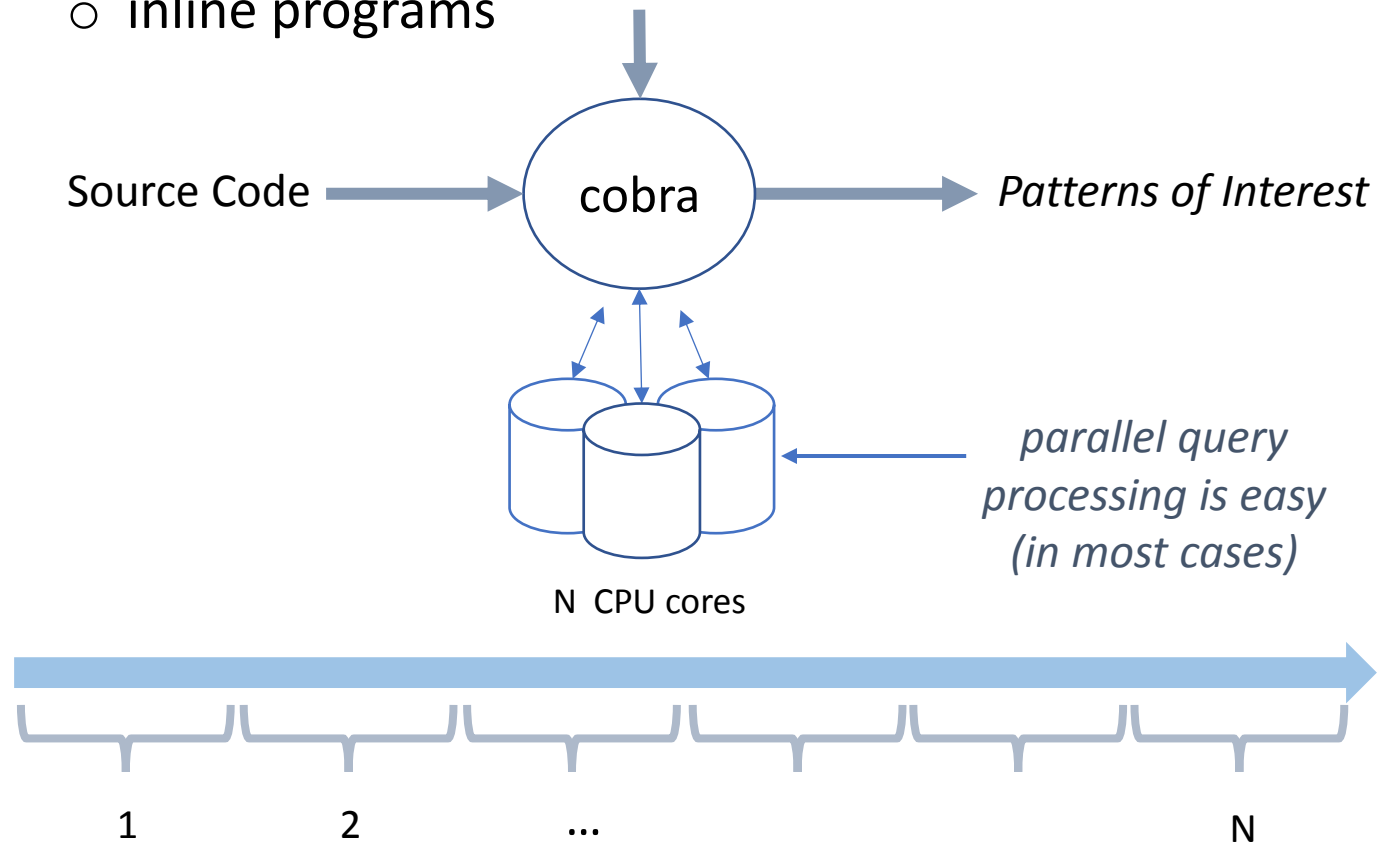


*a linked list of lexical tokens with annotations  
(token types, ranges, levels of nesting for parentheses, brackets, and braces, etc.)*

# cobra's design

minimizing query response time

- interactive query commands over sets & ranges
- pattern matching commands
- inline programs



# getting started

## installation and configuration

```
$ # pick the directory where you'll install the cobra files
$ git clone https://github.com/nimble-code/Cobra
$ ls -l
drwxrwxr-x 2 gh gh 4096 May 16 12:59 bin_linux      # executables for linux
drwxrwxr-x 2 gh gh 4096 May 16 12:59 bin_cygwin    # executables for cygwin
drwxrwxr-x 2 gh gh 4096 May 16 12:59 bin_mac       # executables for macs
drwxrwxr-x 2 gh gh 4096 May 16 10:03 doc          # change history, manpage, license
drwxrwxr-x 2 gh gh 4096 May 16 10:03 gui          # optional small tcl/tk script
drwxrwxr-x 8 gh gh 4096 May 16 15:55 rules        # cobra checker libraries
drwxrwxr-x 1 gh gh 4096 May 16 12:43 src          # cobra source files
drwxrwxr-x 1 gh gh 4096 May 16 12:43 src_app      # standalone cobra checkers

$ cd src
$ sudo make install_mac          # or install_cygwin, install_linux
$ cd ..
$ export PATH=$PATH:`pwd`/bin_mac # or bin_cygwin, bin_linux
$ cobra --configure `pwd`
```

recommended:

install also tcl/tk (wish)  
and graphviz (dot/dotty)

on ubuntu:

```
sudo apt-get install graphviz
```

on mac:

```
brew install graphviz
```

*optional,  
to compile from scratch*

# getting started

all manual pages are online

<http://spinroot.com/cobra>

## Cobra Static Code Analyzer



---

[about](#)

[papers](#)

[manpages](#)

[downloads](#)

---

Cobra is a structural source code analyzer, fast enough that it can be used interactively. The tool prototype (Version 1.0) was developed at NASA's Jet Propulsion Laboratory late 2015, and released for general distribution about a year later.

Versions 2 and 3 of the tool are extended versions that can handle interactive analyses of code bases with up to millions of lines of code, while supporting a significantly richer online query scripting language. It also comes with multi-core support for many types of queries, including a new set of cyber-security related checks.

Starting with Version 3, the Cobra code is distributed in open source form at [github.com/nimble-code](https://github.com/nimble-code).

Cobra can analyze C, C++, Ada, and Python, and can relatively easily be retargeted for other languages. The distribution includes sample query libraries and scripts.

For bug reports and additional information:  
[gholzmann@tsign.acm dot org](mailto:gholzmann@tsign.acm dot org)

---

# getting started

all manual pages are online

<http://spinroot.com/cobra/manual.html>

## COBRA Reference Manual Code Browser and Analysis Tool

---

### Principle of Operation

---

Cobra uses a lexical analyzer to scan in the source code in the files given as arguments on the command-line. It then builds a data structure that can be used for querying that source code, either interactively or with predefined scripts.

The internal data structure that Cobra builds is a basic linked list of lexical tokens, annotated with some basic information and links to other tokens, for instance to identify matching pairs of parentheses, brackets and braces. The tool does not attempt to parse the code, which means that it can handle a broad range of possible inputs. Despite the simplicity of the data structure, the tool can be remarkably powerful in quickly locating complex patterns in a code base to assist in peer review, code development, or structural code analysis.

There are several ways to write queries. You can use:

- Interactive queries (overview below, or see the [index](#)),
- Inline [programs](#) (described separately),
- Standalone [checkers](#) (described separately).

Interactive queries are written in a simple command language that can support the most frequent types of searches. When more complex queries need to be handled, requiring anything other than a sequential scan of the

# getting started

all manual pages are online

<http://spinroot.com/cobra/commands/index.html>

## Cobra Command Overview

### Commands with short-hand:

[a](#) append a source file  
[b](#) move marks back one token  
[B](#) browse a source file (cf V)  
[:](#) (colon) execute a named script  
[c](#) contains: query a range  
[d](#) display  
[.](#) (dot) read a command file  
[e](#) extend match  
[E](#) list of open files  
[G](#) grep in source files  
[?](#) help  
[h](#) command history  
[≡](#) print something  
[i](#) inspect lexical tokens  
[j](#) jump  
[l](#) list  
[m](#) mark tokens

### Commands without short-hand:

[cfg](#) cfg  
[context](#) context  
[cpp](#) preprocessing  
[def...end](#) define named scripts  
[default](#) default  
[fcg](#) fcg  
[fcts](#) fcts  
[ff](#) ff  
[ft](#) ft  
[map](#) map  
[ncore](#) set nr of cores to use  
[nowindow](#) disable window popups for display commands (default)  
[%{...%}](#) inline programs  
[pat](#) pattern token expression  
[pe](#) same as pat  
[re](#) regular token expression



# getting started

all manual pages are online

<http://spinroot.com/cobra/commands/mark.html>

**Cobra**

**Interactive Query Language**

**mark**

## NAME

mark — mark tokens if they match one or two patterns

## SYNTAX

```
m[ark] [qualifier]* pattern [pattern2]
pattern:  string | @string | /re | (expr)
qualifier: ir | no | &
```

## DESCRIPTION

If used without qualifiers, the mark command can only add additional marks, but not remove them. The qualifiers can be used to restrict an existing set of marks to a subset.

A pattern can be one of the following:

- a string (without quotes) to match the token text precisely,
- a token type (when prefixed with a @ symbol),
- a regular expression (when preceded by a / symbol), or
- a pattern expression (when enclosed in round braces).

A qualifier is one of the three terms **ir**, **no**, or **&**. Qualifiers can be escaped as `\no`, `\&`, or `\ir` if a literal match is intended, as can the `/` that would otherwise identify a regular expression, or a round brace `{` that would otherwise indicate a pattern expression.

# getting started

the query libraries

try, for instance:

```
$ cobra -f basic *.c
```

or for summary output:

```
$ cobra -terse -f basic *.c
```

```
$ cd $COBRA/rules
$ ls -l
total 60
drwxr-xr-x+ 1 gh None 0 May  1 16:31 cwe
drwxr-xr-x+ 1 gh None 0 Oct 11 2018 jpl
drwxr-xr-x+ 1 gh None 0 May  6 17:16 main
drwxr-xr-x+ 1 gh None 0 Oct 11 2018 misra
drwxr-xr-x+ 1 gh None 0 Oct 11 2018 pedantic
drwxr-xr-x+ 1 gh None 0 Jun  1 14:18 play
drwxr-xr-x+ 1 gh None 0 Mar 20 15:49 stats
$ ls -l main/*.cobra
total 89
-rwxr-xr-x+ 1 USER None 1017 May 12 2017 basic.cobra
-rwxr-xr-x+ 1 USER None 3513 May 13 2017 binop.cobra
-rwxr-xr-x+ 1 USER None  21 May  6 17:16 cwe.cobra
-rwxr-xr-x+ 1 USER None  793 Apr 20 2017 extern.cobra
-rwxr-xr-x+ 1 USER None 2490 May 13 2017 iridex.cobra
-rwxr-xr-x+ 1 USER None 4004 May 15 2017 jpl.cobra
-rwxr-xr-x+ 1 USER None  589 May 12 2017 metrics.cobra
-rwxr-xr-x+ 1 USER None  714 May 12 2017 misra1997.cobra
-rwxr-xr-x+ 1 USER None  725 May 12 2017 misra2004.cobra
-rwxr-xr-x+ 1 USER None  658 May 12 2017 misra2012.cobra
-rwxr-xr-x+ 1 USER None  501 May 12 2017 p10.cobra
-rwxr-xr-x+ 1 USER None 1008 May 31 09:02 reverse_null.cobra
-rwxr-xr-x+ 1 USER None  585 May  6 17:09 stats.cobra
```

# getting started

## languages supported

- Cobra is designed to be language neutral, which means that:
  - it can be targeted to a broad range of languages, by providing it with the relevant set of lexical tokens
  - the token categories for C, C++, Java, Ada, and Python are predefined
  - the default is C, the alternatives:
    - \$ cobra -Ada ...
    - \$ cobra -Java ...
    - \$ cobra -C++ ...
    - \$ cobra -Python ...
  - other languages can be added by using the *map* command
- to see all currently recognized cobra flags:
  - \$ cobra --

pattern search

# pattern search

so what's wrong with using "grep"?

```
$ grep -e x *.c | wc  
1136      7251      57700
```

```
sample match: prefix = s;
```

```
$ cobra -pat x *.c | wc  
96        549        3647
```

matches *tokens* named *x*,

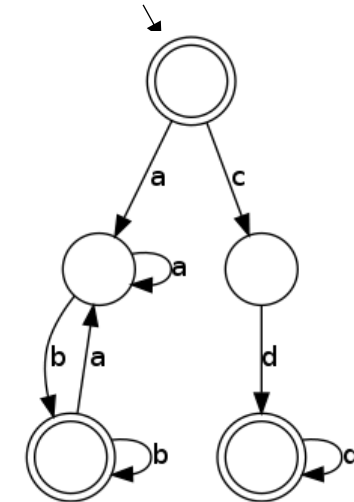
```
sample match: strcmp(x->txt, "x")
```

*note: the pattern search does not match either the word prefix or the string "x"*

# pattern search

pattern expressions are a simplified form of regular expressions

- Regular expressions are used in many tools and applications for pattern matching *text strings*
- Examples include the well-known Unix™ tools:
  - grep, sed, awk, lex, ed, sam, etc.
  - Google search patterns can also contain regular expressions
- Regular expressions define *finite state automata*
  - the automata accept precisely those text strings that match the regular expression
  - example: "(a+ b+)\* | c d+" defines the finite state automaton (FSA) shown on the right.
    - the FSA *accepts* input if it terminates in an accepting state (indicated by the double circles)



(, ), +, \*, and | are regular expression meta-symbols)

# pattern search

cobra patterns are defined over *lexical tokens* instead of *text-strings*

```
$ cobra -pat x *. [ch] # a very simple 'pattern'
```

```
$ cobra -pat '{ .* malloc ^free* }' *.c # don't-cares, negation, repetition
```

*cobra guarantees that in all these patterns  
the nesting level of all brace pairs matches*

```
$ cobra -pat '{ .* [static STATIC] .* }' *.c # choice
```

```
$ cobra -pat '{ .* @type x:@ident ^:x* }' *.c # types and name-binding
```

```
$ cobra -pat 'x:@ident -> .* if ( :x /= NULL )' *.c # /regex embedded
```

without spaces:  
this matches a single token

think about this one....  
to match the *token* /= write: `\/=`

# pattern search

matching for-loops not followed by a compound statement

```
$ cobra -pat `for ( .* ) ^{ ` *.c # first try
5814     for (n = v_names[ix]; n; lastn = n, n = n->nxt) // mk_var
5815     {
           if (n->h2 > h2)

$ cobra -pat `for ( .* ) ^[{ @cmnt]*' *.c # second try
2834     for (i = 0; i < Ncore; i++)
2835     for (n = a_tbl[i].n[h1]; n; n = n->nxt) // sum_array

$ cobra -pat `for ( .* ) ^[{ @cmnt for switch if]*' *.c # third try
793     for (yylen = 0; yystr[yylen]; yylen++)
794         continue;
```

# pattern search

adding preprocessing with -cpp

```
$ cobra -cpp -pat 'for ( .* ) ^{' *.c
```

# pattern search

adding preprocessing with -cpp

```
$ cobra -cpp -pat 'for ( .* ) ^{' *.c
```

```
cobra_ctok.c:
```

```
1: 2483 for ( i = 0; i < nr; ++i )
```

```
2: 2484     *(dst++) = *(src++);
```

```
3: 2538 YY_INPUT(( & ... ), ...
```

```
...
```

?

# pattern search

adding preprocessing with -cpp

```
$ cobra -cpp -pat 'for ( .* ) ^{' *.c  
cobra_ctok.c:  
1: 2483 for ( i = 0; i < nr; ++i )  
2: 2484 *(dst++) = *(src++);  
  
3: 2538 YY_INPUT(( & ... ), ...  
...
```

*macro expansion*

```
...  
for (n=0; n < max_size && \  
    (c = getc(yyin)) != EOF && c != '\n'; ++n) \  
    buf[n] = (char) c; \  
...
```

# traditional regular expressions are also supported, but rarely needed

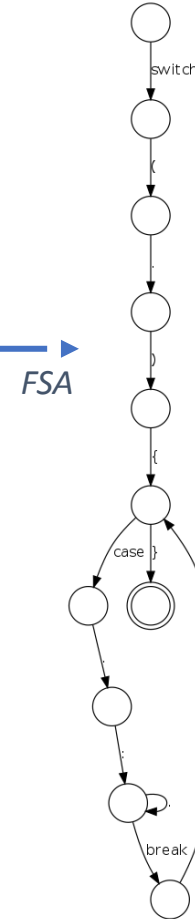
*note the required spaces to separate tokens*

- example

\$ cobra **-regex** 'switch \( . \) { ( case . : .\* break ; ) \* }' \*.c

( and ) are now *meta-symbols* (used for grouping)  
as are +, ?, and |

a plain ( or ) must now be written \*(* and \*)* to distinguish  
them from the *meta-symbols*



# regular expressions vs pattern expressions

## the key differences

( and )	grouping
	choice, e.g. “(a   b)” matches a <i>or</i> b
+	one or more repetitions
?	zero or <i>one</i> repetition
*	zero or more repetitions
.	match any token
@type	match a particular token class, e.g., @ident
x:@type	bind the variable-name <b>x</b> to a specific token <i>name</i>
:x	refer to a previously bound name
[ and ]	define a set of options, e.g., [a b c] matches one of a b or c

*not meta-symbols in  
pattern expressions*

* and ]	when <i>preceded</i> by a space is a regular symbol
[	when <i>followed</i> by a space is a regular symbol
/re	match token if the <i>token-text</i> matches re

*in pattern  
expressions*

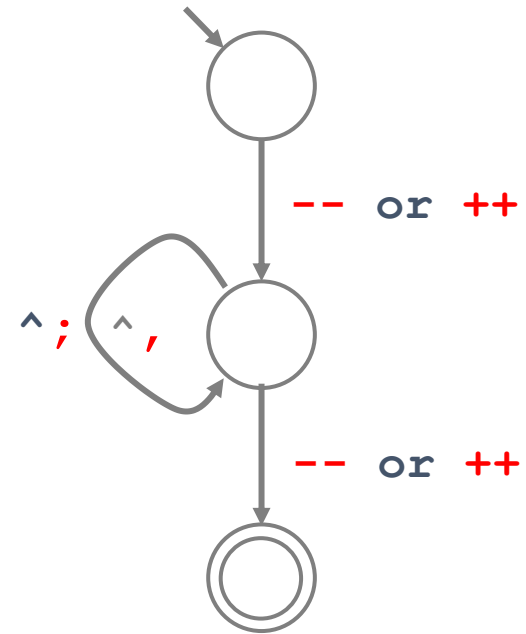
# pattern search

find multiple side-effects without a *sequence point* in between

```
$ cobra -pat '[-- ++] ^[; ,]* [-- ++]' *.c
```

```
sml_dsa.c:
```

```
16: 212 for (; k >=0; k--) {  
17: 213     lA1 = (ulong) (*p++)*(*q--);  
18: 230 for (; j < len; j++) {  
19: 231     lA1 = (ulong) (*p++)*(*q--);
```



# pattern search

## interactive use of pattern queries

```
$ cobra -N8 `cat thousands_of_filenames` # e.g., linux-4.3
```

```
8 cores 39133 files 84,111,645 tokens
```

```
: # if/else/if chains must end with else
```

```
: pat else if ( .* ) { .* } ^else
```



*matched braces*

```
: # every non-void fct must have a return stmt
```

```
: pat ^void @ident ( .* ) { ^return* }
```

no quotes around inline  
patterns are required  
but now a ";" token match  
must be protected with "\;"  
to prevent interpretation as  
a query-command separator

# pattern search

interactive use of pattern queries

- : # check the sanity of for-statements
- : **pat for** ( **.\*** \; **.\*** [**<** **<=**] **.\*** \; **.\*** **^[++ +=]** )  
memtest.c: 108: # linux-4.3  
    for (i = memtest\_pattern-1; i < UINT\_MAX; --i) {  
timeconv.c:120  
    for (y = 11; days < ip[y]; y--)
- : # or with regular expressions on token *texts*
- : **pat for** ( **.\*** **/^<** **.\*** **/^- [-=]** **\$** **.\*** )

# pattern search

using name binding

# find assignments to the control variable of a for-loop, inside the loop body

```
$ cobra -pat "for ( x:@ident .* ) { .* :x = .* }" *.c
```

*matching braces*

*matching braces*

# find local variable declarations that aren't used in the function body

```
$ cobra -pat ") { .* @type x:ident ^:x* }" *.c
```

*to avoid matching  
on structure declarations*

note: *all* individual *tokens* in the pattern  
must be separated by *spaces*

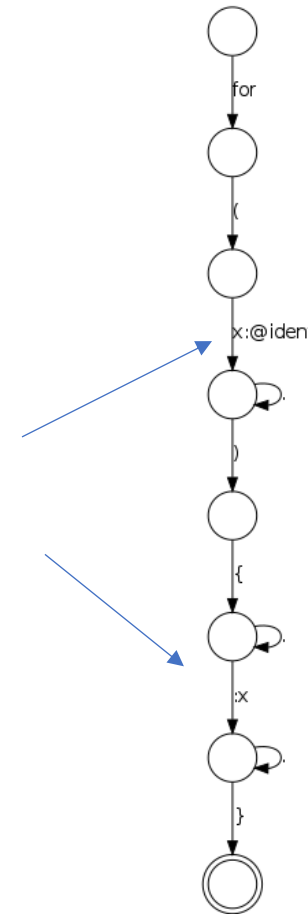
# pattern search

find uses of the control variable of a for-statement inside the body of the loop, using variable binding

```
$ cobra *.c
: pat for ( x:@ident .* ) { .* :x .* }
program.c:
1: 37 for (i = 0; i < 10; i++)
2: 38 { i++;
3: 39 }
4: 48 for (i = 0; i < 10; i++)
5: 49 { stmt6();
6: 50 i = 12;
7: 51 }
```

*name  
binding &  
reference*

*Cobra converts the pattern into an NFA, converts that into a minimized DFA, and uses that to perform the search*



# pattern search

## the matching algorithm

**Regular Expression Matching Can Be Simple And Fast**  
(but is slow in Java, Perl, PHP, Python, Ruby, ...)

[Russ Cox](#)

[rsc@swtch.com](mailto:rsc@swtch.com)

January 2007



### Introduction

This is a tale of two approaches to regular expression matching. One of them is in widespread use in the standard interpreters for many languages, including Perl. The other is used only in a few places, notably most implementations of awk and grep. The two approaches have wildly different performance characteristics:

Time to match  $a^n a^n$  against  $a^n$

# pattern search

## the matching algorithm

Regular Expression Matching Can Be Simple And Fast  
(but is slow in Java, Perl, PHP, Python, Ruby, ...)

[Russ Cox](#)

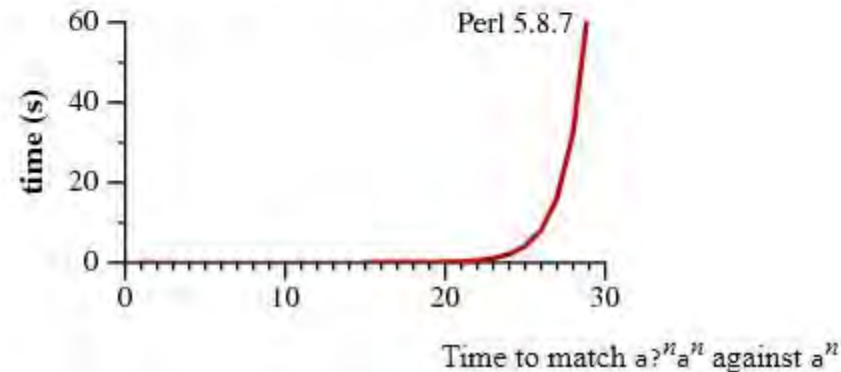
[rsc@swtch.com](mailto:rsc@swtch.com)

January 2007



### Introduction

This is a tale of two approaches to regular expression matching. One of them is in widespread use in the standard interpreters for many languages, including Perl. The other is used only in a few places, notably most implementations of awk and grep. The two approaches have wildly different performance characteristics:



# pattern search

## the matching algorithm

Regular Expression Matching Can Be Simple And Fast  
(but is slow in Java, Perl, PHP, Python, Ruby, ...)

[Russ Cox](#)

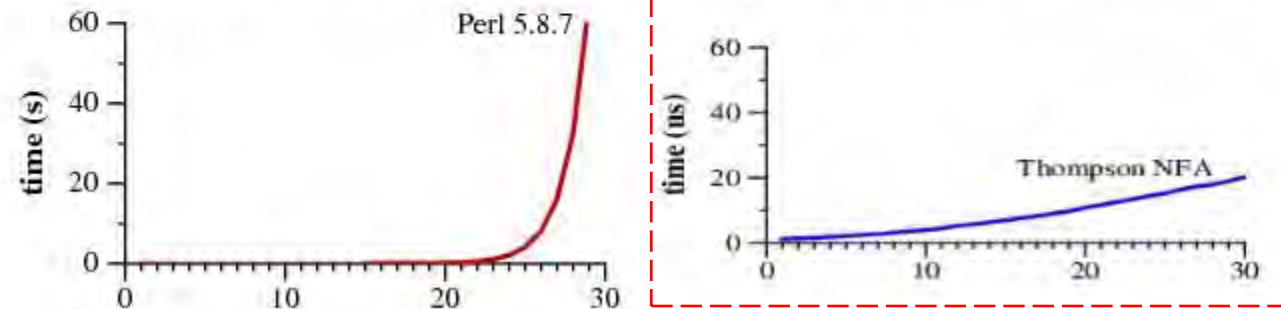
[rsc@swtch.com](mailto:rsc@swtch.com)

January 2007



### Introduction

This is a tale of two approaches to regular expression matching. One of them is in widespread use in the standard interpreters for many languages, including Perl. The other is used only in a few places, notably most implementations of awk and grep. The two approaches have wildly different performance characteristics:



Time to match  $a^n a^n$  against  $a^n$

# pattern search

thompson's algorithm



# pattern search

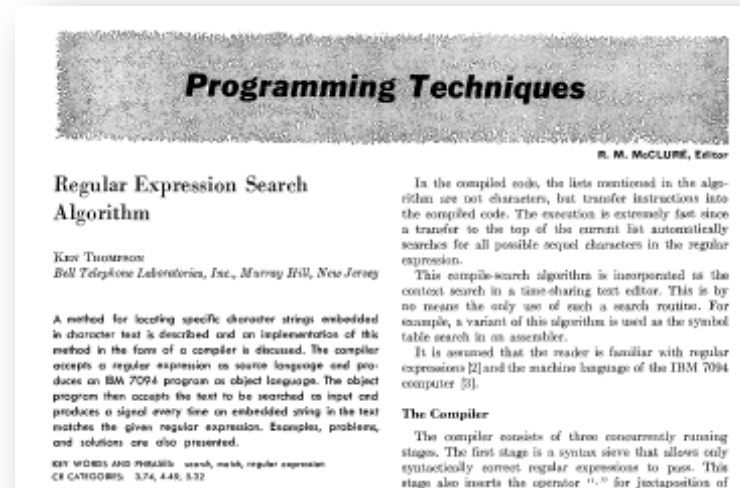
thompson's algorithm

example: find expressions with multiple side-effects

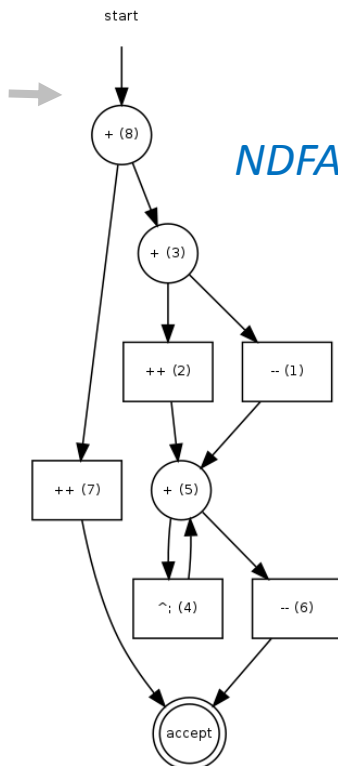
```
$ cobra -pat "[ -- ++ ] ^ ; * [ -- ++ ]" *.c
```

```
sml_dsa.c:
```

```
213 IA1 = (ulong) (*p++)>(*q--);
```



(CACM 11:6 1968)



```
$ cobra -view -pat '...' *.c  
(requires graphviz and x11)
```

examples

# pattern search

## examples of pattern searches

: pat @type /restore ( .\* ) { .\* = false .\* }

# find function names containing “restore”

: pat x:@ident += sprintf ( ^,\* :x .\* /%s .\* )

# using result of sprintf in first arg

: pat /define @ident ( x:@ident ) ^[EOL ( ]\* :x

# macro args must be enclosed in braces

: pat typedef ^\;\* /\_ \;

# there can be no \_ in a typedef names

: pat typedef ^\;\* \* ^\;\* ^/\_ptr \;

# typedefs for ptrs must have a \_ptr suffix

: pat x:@ident -> .\* if ( :x / = NULL )

# no ptr dereferencing preceding a NULL check

: pat if ( .\* ) ^{

# body of if statement not enclosed in { ... }

interactive queries

# the query language

## overview

there are about 40 query commands predefined,  
though 4 or 5 suffice for handling most queries.  
they can be used for:

examples:

- A: mark, next, back, jump, contains, extend, undo, reset
- B: stretch
- C: display, list, pre, =, help
- D: history (h), browse (B), files (F), system (!), cfg, fcg, fcts
- E: save (>), restore (<)

- A. Setting, Moving, or Removing Marks
- B. Setting Ranges
- C. Output
- D. Meta Commands
- E. Defining Sets of Marks

try:

```
$ cobra -c help < /dev/null  
and note that the output is different from:  
$ cobra -help
```

# the query language

an example of an interactive session: find switch statements without a default clause

```
$ cobra *.ch          # start an interactive session on the cobra 3.0 sources
1 core, 13 files, 58381 tokens
: mark switch (      # mark all switch statements
29 matches
: next {             # move mark to the start of the body
29 matches
: contains no default # check the range from { to }, no is a qualifier
6 matches
: display 2 +8       # display the 2nd match with 8 lines after it
cobra_lib.c:538:
2: > 538  {   switch (*s) {
2: 539      case '&':
2: 540      case '|':
2: 541      case '^':
2: 542          tmp = t;
2: 543          t = s;
2: 544          s = tmp;
2: 545          break;
2: 546  }   }
: quit
$
```

# the query language

## shorthands

instead of writing:

```
: mark switch (  
: next {  
: contains no default  
: display
```

we can also use shorthands:

```
: m switch (  
: n {  
: c no default  
: d
```

m[ark]	defines a set of matches
n[ext]	moves all current marks forward
d[isplay]	displays the current marks
c[ontains]	checks a token range
{ ... }	defines a token range, as do:
[ ... ]	
( ... )	

and we can combine commands on a single line:

```
: m switch (; n {; c no default; d
```

or execute everything from *the command line*:

```
$ cobra -c 'm switch (; n {; c no default; d' *.c
```

# the query language

## command-line use

```
$ cobra -c "m switch (; n {; c top no default; d" *.*[ch]
```

```
478     switch (f->n->ntyp) {
479     case UNLESS:
480         attach_escape(f->sub->this, e);
481         break;
482     case IF:
483     case DO:
484         for (z = f->sub; z; z = z->nxt)
485             attach_escape(z->this, e);
486         break;
487     case D_STEP:
488         /* attach only to the guard stmt */
489         escape_el(f->n->sl->this->frst, e);
490         break;
491     case ATOMIC:
492     case NON_ATOMIC:
493         /* attach to all stmts */
494         attach_escape(f->n->sl->this, e);
495         break;
496     }
```

another query *qualifier* to restrict the check to the *top level* of nesting

```
# with a -runtimes flag (and without the 'd'):
$ cobra -runtimes -c 'm switch; n {; c top no default' *.*c
(0.0404 sec)
(0.00338 sec)
(0.000523 sec)
(0.000344 sec)
(0.0005 sec)
$
```

# the query language

## the stretch command

we've already covered *five* commands: `mark`, `next`, `contains`, `display`, and `quit`  
three other useful commands are `stretch`, `list`, and `pre`:

```
$ cobra *.c
1 core, 10 files, 56450 tokens
: mark for (                # mark all for statements
206 matches
: next \;                   # move mark to the first ; after for
206 matches
: stretch \;               # define a range from here to the next semi-colon
206 matches
: contains ->               # restrict to ranges that contain a -> token
45 matches
: pre 1                     # show the first matched range with pre (or p)
cobra_cfg.c:38<->cobra_cfg.c:38
 1:      38  for ( cur = ( Prim * ) n ; rval && cur && cur -> seq <= n -> jmp -> seq ; cur = cur -> nxt )
 1:                                ^  ^^^^  ^^  ^^^  ^^  ^^^  ^^  ^^^  ^^  ^  ^^  ^^^  ^^  ^^^  ^
```

try using `display (d)`, or `list (l)` commands instead of `pre (p)`

# the query language

## command qualifiers

we've used two qualifiers so far: `top` and `no`

there are two more: `&` and `ir`

`top` # restrict to matching at the same nesting level as the mark (`contains` and `stretch`)

`no` # to find non-matches (`mark` and `contains`)

`ir` # mark *all* matching tokens inside the current range (`mark`)

`&` # restrict to *marks* that also match a new pattern (`mark`)

`&` # restrict to *ranges* that also match a new pattern, and move the mark to *the first* (`contains`)

to see how it works, at the end of the last example, type:

```
: c & seq
```

```
: p 1
```

```
cobra_cfg.c:38:
```

```
1:      38  for ( cur = ( Prim * ) n ; rval && cur && cur -> seq <= n -> jmp -> seq ; cur = cur -> nxt )  
1:                                     ^^^
```

try instead: `m ir seq`

note: only the first match is marked

# the query language

find if-else-if chains not ending with 'else' (another MISRA rule violation)

we earlier expressed this with a search

pattern: `else if ( .* ) { .* } ^else`

we can also do it with a

sequence of query commands

```
$ cobra *.c
: mark else if
40 matches
: next (
40 matches
: jump
40 matches
: next
40 matches
: mark & {
33 matches
: jump
33 matches
: next
33 matches
: mark no else
11 matches
: display 1 -5
```

# to the other end of the range

# restrict the set of matches

```
cobra_fcg.c:151:
1: 146          }
1: 147          } else if (r->curly > 0)
1: 148          {   continue;
1: 149          }
1: 150
1: > 151        ptr = r;
```

try: adding `d[isplay]` or `l[ist]` commands to see which tokens are matched at different steps, e.g., `list 1 2`

the 1-liner:

```
: m else if; n (; j; n; m & {; j; n; m no else; d 1 -5
```

examples

# the query language

## examples of command queries

- 1: find global variables with fewer than 3 characters
- 2: find loops that contain gotos but no labels
- 3: find recursive functions
- 4: find goto statements immediately followed by the label

### answer 1

```
: m @ident  
: m & (.curly == 0 && .round == 0 && .len < 3)
```

### answer 2

```
: for; m do; m while  
: n (; j; n; m & {  
: c goto; c no :
```

### answer 3

```
: fcts          # mark all fct names  
: s top }       # stretch to end of fct body  
: m ir $$       # the fct name is bound to $$  
: n; m & (; b   # make sure these are all fct calls
```

```
.range  # nr of lines in a range  
.fnm    # source filename (a string)  
.lnr    # source line-number  
.curly  # level of {...} nesting  
.round  # level of (...) nesting  
.bracket # level of [...] nesting  
.len    # length of token text  
.typ    # token type (a string)  
.txt    # token text (a string)  
.seq    # token sequence number  
.mark   # marked value
```

### answer 4

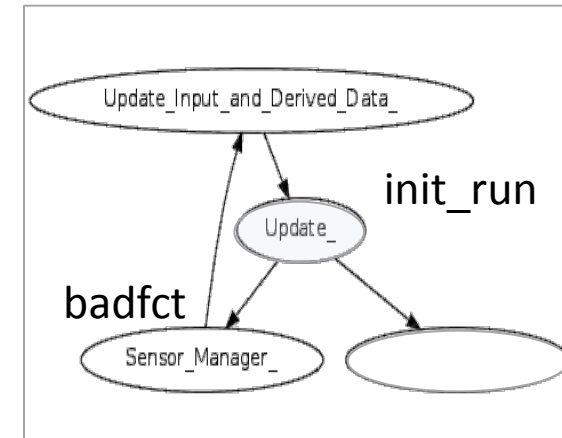
```
: pat goto x:@ident \; :x  
: m goto; n; s $$; c no \;
```

# the query language

using sets

is there memory allocation ever used after system initialization in C++ code?

```
$ cobra -C++ *.cpp
: fcts                # mark all fct names
: next {             # move to fct body
: contains new       # restrict to these
: back (            # back to fct parameter list
: back              # back to fct name
: >1                # store these names in set 1
: reset             # clear all marks
: fcg init_run *    # mark all fcts reachable from init_run
: <&1               # check intersection with set 1
: fcg init_run badfct # show call graph connecting init_run and badfct
```



# the query language

operations on sets, and the use of set qualifiers

- `>n` # save all current marks and ranges in set n
- `<n` # restore current marks and ranges from set n
- `<|n` # add marks from set n to current (set union)
- `<&n` # keep only marks also in set n (set intersection)
- `<^n` # keep only marks not in set n (set difference)

where n is 1..3

two additional sets are used internally for  
storing the current and the previous set of marks  
(allowing a fast 'undo' on most operations)

# the query language

ranges: stretch and extend

the *stretch* command can refer to the current token name as \$\$

example, MISRA 2012 rule 2.7: *there should be no unused parameters in function declarations*

```
$ cobra -cpp *.c # preprocessed cobra sources
1 core, 10 files, 56450 tokens
: fcts # mark all function definitions (at the names)
: n ( # move mark to start of parameter list
: m ir @ident # mark all identifiers in the list (c & @ident would just mark the first)
: e /^[,)]$ # retain only those marks that are followed by , or ) (the extend command)
: >1 # store these names in set 1
: s $$ # try to stretch each marked identifier to the next occurrence
: >2 # store the names for which this succeeds in set 2
: <1 # recover set 1
: <^2 # omit all the ones also stored in set 2
4 matches
: = "misra r2.7: there should be no unused parameters:"
: d
```

# pattern searches

stretch and extend: find unused parameters

```
4 matches
: d
cobra_heap.c:313:
  1:   313  stop_timer(int cid, int always, const char *s) ?
cobra_lib.c:2900:
  2:   2900  history(char *unused1, char *unused2)
  3:   3209  cleanup(int unused)
: d 1 +30
cobra_heap.c:313:
  1: >  313  stop_timer(int cid, int always, const char *s)
  1:   ...
  1:   326  #if 0
  1:   327      if (always
  1:   328      || delta_time[cid] > 0.1)
  1:   329  #endif
  1:   330      { if (Ncore > 1)
  1:   331          { printf("%d: ", cid);
  1:   332          }
  1:   ...
```

# the query language

## defining functions

```
: def p10_rule4(rn, nr)
  fcts          # mark function names
  n {          # move to fct body
  m & (.range > nr) # restrict to fcts longer than nr lines
  b
  = "=== rn: functions exceeding 75 physical source lines:"
  d
end
: p10_rule4(R4, 75)
```

*.range* is a predefined token attribute

*token attributes* that can be referenced:

<i>.fnm</i>	# source filename (a string)
<i>.lnr</i>	# source line-number
<i>.curly</i>	# level of {...} nesting
<i>.round</i>	# level of (...) nesting
<i>.bracket</i>	# level of [...] nesting
<i>.len</i>	# length of token text
<i>.typ</i>	# token type (a string)
<i>.txt</i>	# token text (a string)
<i>.seq</i>	# token sequence number
<i>.mark</i>	# marked value

# the query language

defining new token types (.typ): map

```
$ cat prepositions.map
```

```
of      preposition
to      preposition
in      preposition
for     preposition
on      preposition
with   preposition
by     preposition
but    preposition
at     preposition
from   preposition
about  preposition
like   preposition
into   preposition
```

```
...
```

```
# map token text to new user-defined token types
```

```
$ cobra *.txt # some random English prose
: map prepositions.map
: m @preposition .
: = "a sentence should not end with a preposition:"
: d
```

# the query language

refining pattern search results using `.mark`

using the `.mark` token attribute  
we can also conveniently refine *pattern*  
searches

by default all tokens in a matched  
pattern are marked with **1**  
there are two special cases:

- the first token in each pattern  
is marked with value **2**
- any bound variables in the pattern  
is marked with value **3**

```
$ cobra *.c
: pat for ( x:@ident .* ) { .* :x = .* }
bound variables matched:
  1: cobra_te.c:1579: q_now
  2: cobra_te.c:1358: m
  3: cobra_te.c:881: b
  4: cobra_sym.c:105: r
  5: cobra_sym.c:51: r
  6: cobra_prep.c:339: c
  7: cobra_lib.c:2492: r
  8: cobra_lib.c:2122: z
  9: cobra_lib.c:1202: s
 10: cobra_lib.c:782: r
 11: cobra_fcg.c:156: r
 12: cobra_cfg.c:171: cur
 13: cobra_cfg.c:67: cur
13 patterns matched
3929 matches
: m & (.mark == 2)
12 matches
: undo
: m & (.mark == 3)
14 matches
```

# the query language

reading commands from files

- for instance, to read a query function from the “play” library use the dot command `.` :

```
$ cobra *.c  
: . play/declarations.cobra
```

- we can do the same from the command line:

```
$ cobra -f play/declarations.cobra *.c
```

- cobra query files stored in rules/main can be read without the directory prefix
- try, for instance:

```
$ cobra -terse -f basic *.c  
$ cobra -terse -f stats *.c  
$ cobra -terse -f metric *.c  
$ cobra -terse -f cwe *.c
```

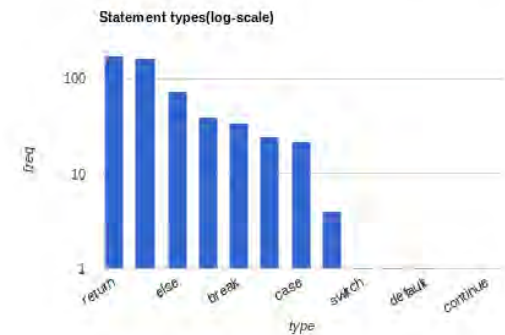
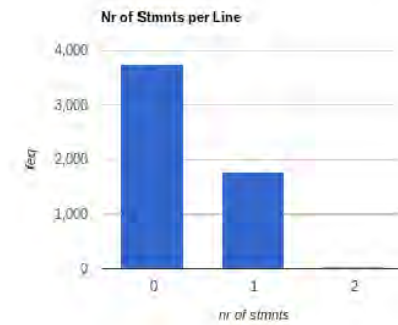
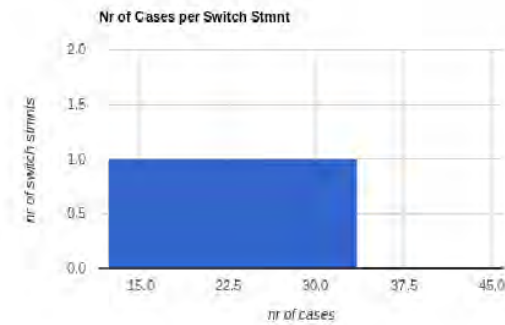
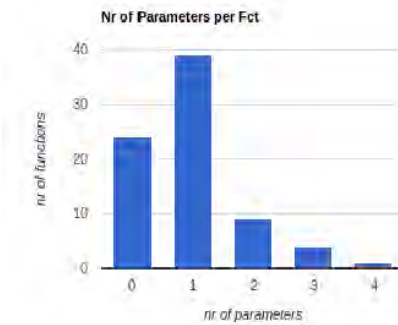
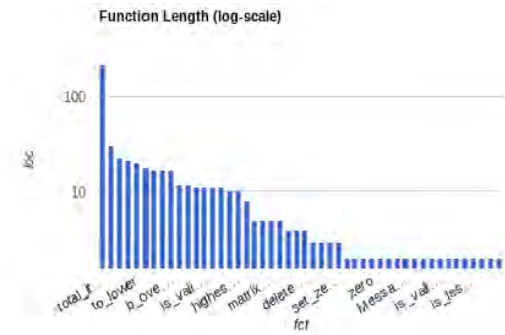
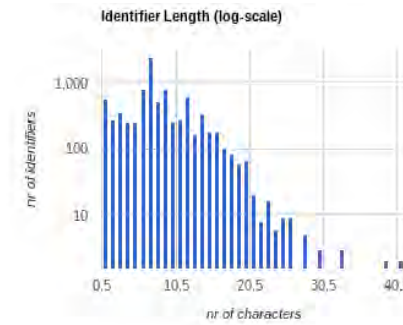
# the query language

generating stats: e.g., tabulate the length of switch statements

```
$ cobra -c 'm switch; n {; = (.range)' *.c | sort -k3 -n  
cobra_prog.c:999      value 3  
cobra_eval.c:1125     value 4  
...  
cobra_lib.c:1030      value 200  
cobra_prog.c:1688     value 437  
cobra_ctok.c:1361     value 1076
```

# the query language

you can now input these data to google graphs to create dash-boards plotting quality metrics of a code base



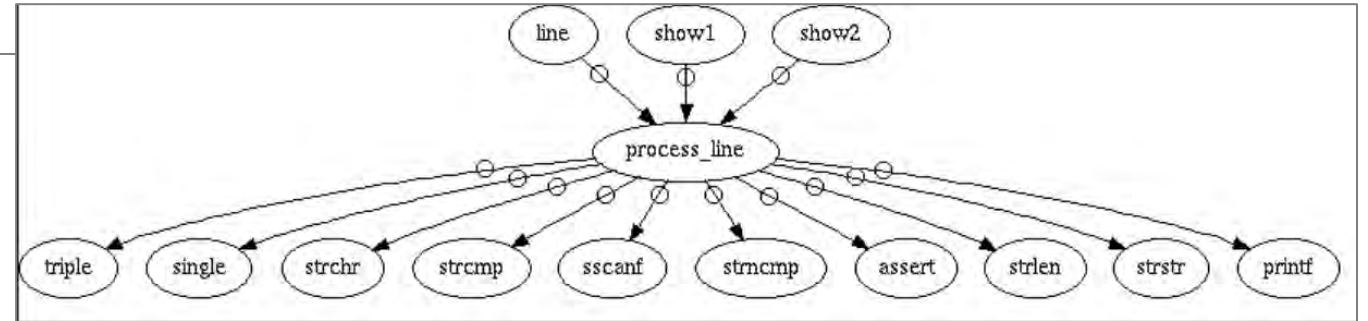
Data (log-scale)

Comments and Preprocessor (log-scale)

# the query language

computing the function call context (requires graphviz/dot)

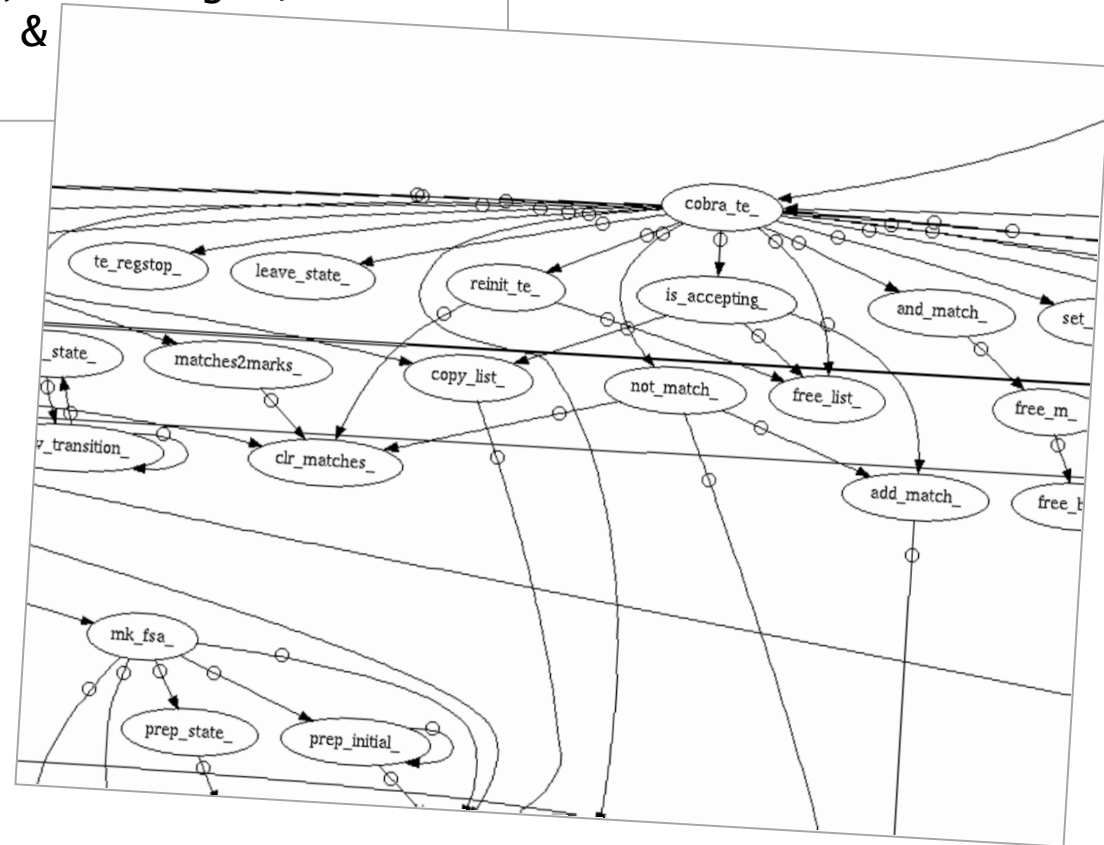
```
$ cobra *.c
1 core, 10 files, 56546 tokens
: context process_line
cobra_prim.c:626-673
calls:
    cobra_prim.c:671: triple()
    cobra_prim.c:669: single()
    cobra_prim.c:662: strchr()
    cobra_prim.c:649: strcmp()
    cobra_prim.c:647: sscanf()
    cobra_prim.c:646: strncmp()
    cobra_prim.c:644: assert()
    cobra_prim.c:640: strlen()
    cobra_prim.c:638: strstr()
    cobra_prim.c:631: printf()
is called by:
    cobra_lex.c:809: line()
    cobra_lex.c:279: show1()
    cobra_lex.c:288: show2()
:
```



# the query language

or full or partial function call graphs (requires graphviz/dot)

```
$ cobra *.c  
1 core, 10 files, 56546 tokens  
: fcg  
wrote: /tmp/cobra_dot_sf8RAV (269 nodes, 499 edges)  
view with: !dotty /tmp/cobra_dot_sf8RAV &  
: !dotty /tmp/cobra_dot_sf8RAV &
```



# the query language

## browsing code

```
: B cobra_te.c 100 # show file starting at line 100
: B                # browse forward in same file
: B 90             # browse same file from line 90
```

with tcl/tk installed:

```
: V cobra_te.c 100 # like B, but in a popup window
: window           # enable automatic window popups
: m while          # mark something
: d 1              # now pops up a tcl/tk window with the source text
: : window off     # disable window popups
```

with graphviz (dotty) installed:

```
$ cobra -view -pat "... " *.c # pop up graph showing FSA for the pattern
```

other sometimes useful short-hands:

```
: ff par_scan      # find function definition
: cpp on           # processes the header files
: ft Prim          # find a type definition
```

# the query language

track\_start, track\_stop, and shell escape

```
$ cobra *.ch
1 core, 15 files, 90063 tokens
: m while
127 matches
: track start file1 # redirect output
: list
: track stop # end redirection
: !wc file1 # shell escape
: q
$
```

# the query language

how does it scale?

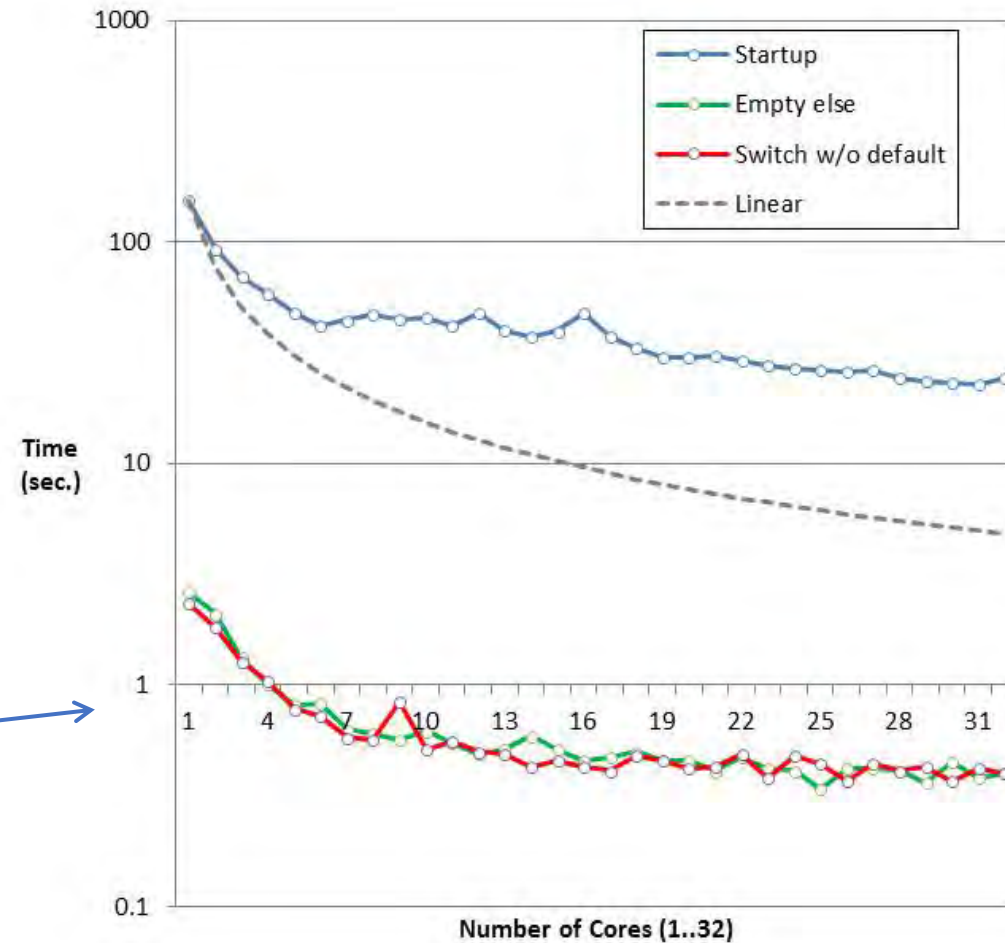
18,633,817 Lines of Code of the Linux 4.3 distribution, with 39,144 .c and .h files

checking 2 types of queries:

- find empty else stmts
- find all switch stmts without default clause

using 1..32 CPU cores

with 4 or more cores we get interactive query processing times < 1 sec.



inline scripting

# cobra inline programs

the scripting language

An inline program is enclosed in delimiters:

```
%{  
    ...  
%}
```

which can be used like any other query command, e.g. in a query function:

```
def prog1  
    %{  
        ...  
    %}  
end
```

and invoked by name:

```
: prog1
```

If stored in a file, these scripts can be invoked from the command line as well:

```
$ cobra -f file.cobra *.[ch]
```

Simple example:

```
$ cat file.cobra  
%{  
    print .fnm ":" .lnr ": " .txt "\n";  
%}
```

which prints the text of all tokens, each preceded by filename and linenumber

# cobra inline programs

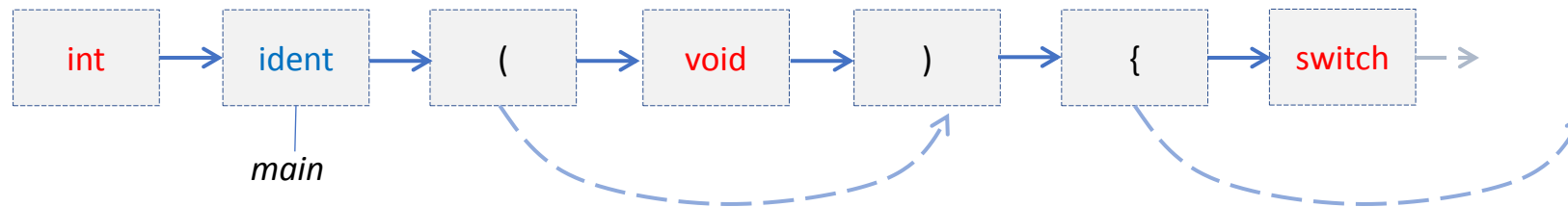
the scripting language

## the main control-loop:

cobra inline programs are, like all other query commands, executed once for each token in the input sequence

but a cobra program can also take over control and navigate the token sequence in any way it wants

it can refer to any token attribute



# cobra inline programs

## keywords

### control flow :

if  
else  
while  
break  
continue  
goto  
for  
in  
return  
Next  
Stop

### token references:

Begin

End

.

example

predefined

functions:

print

assert()

user-defined

functions:

function name (...) { ... }

### a fairly standard grammar:

```
if (expr) { stmt;+ } [ else { stmt;+ } ]
```

```
while (expr) { stmt;+ }
```

```
for (var in array) { stmt;+ }
```

```
L: goto L;
```

```
q = .;
```

```
array[expr] = expr;
```

### variables:

can be introduced without declaration  
the type is inferred from context, and  
may change dynamically

# cobra inline programs

two very simple examples, using the predefined function *print*

```
%{  
    print .fnm ":" .lnr ": " .txt "\n";  
%}
```

```
%{  
    print "hello world\n";  
    Stop;    # no need to repeat this for every token..  
%}
```

# cobra inline programs

token attributes that can be referred to and modified

the *current* token is always referred to as dot: . <http://spinroot.com/cobra/commands/tokens.html>

integer values:

.round  
.bracket  
.curly  
.len  
.lnr  
.mark  
.seq  
.range

string values:

.fct  
.fnm  
.txt  
.typ

token positions:

.  
.nxt  
.prv  
.jmp  
.bound

reference rule:

token attributes can only be referenced directly, so instead of writing:

```
q = .nxt.jmp;
```

you have to split this in two steps:

```
q = .nxt; q = q.jmp;
```

example-1 (using a variable *q*):

```
q = .nxt;  
q.mark++;  
q.fnm = "hello";  
q = q.prv;
```

example-2:

```
if (.txt == "{")  
{  
    q = .jmp;  
    assert(q.seq != 0);  
    r = q.jmp;  
    assert(r == .);  
}
```

# cobra inline programs

operators that can be used in expressions

binary operators:

<code>+, -, *, /, %</code>	arithmetic (integer operands)
<code>&gt;, &lt;, &lt;=, &gt;=, ==, !=,   , &amp;&amp;</code>	Boolean (operands can be any type)

unary operators

<code>!</code>	Boolean, logical negation	
<code>-</code>	arithmetic, unary minus	
<code>~</code>	true if .txt contains a pattern	<code>if (~yy) { ... }</code>
<code>^</code>	true if .txt starts with a pattern	<code>if (^yy) { ... }</code>
<code>#</code>	true if .txt equals a pattern	<code>if (#yy) { ... }</code>
<code>@</code>	true if .typ equals a pattern	<code>if (@ident) { ... }</code>

regular expression matching of any token text:

`match(s1, s2)` true if s1 matches s2, where s2 can be a regex  
`if (match(q.txt, "[Yy][Yy]")) { ... }`

comments: `#` when followed by another `#` or a space

# cobra inline programs

associative arrays (“hash-maps”)

name [ expr [, expr]\* ]      associates a (possible sequence of) values, or any type with another value, of any type (a “map”)

predefined functions for associative arrays:

retrieve(A, n)

retrieves the nth element of associative array A

size(A)

returns the number of elements stored in array A

unset A[v]

remove associative array element A[v]

unset A

remove variable or array A

# cobra inline programs

## example 1

find the 10 most frequently occurring trigrams of types

```
%{
    q = .nxt;
    r = q.nxt;
    if (.typ != "" && q.typ != "" && r.typ != "")
    {   Trigram[.typ, q.typ, r.typ]++;
    }
}%
track start _tmp_
%{
    for (i in Trigram)
    {   print i.txt "\t" Trigram[i.txt] "\n";
    }
    Stop;
}%
track stop
!sort -k2 -n < _tmp_ | tail -10; rm -f _tmp_
%{ unset Trigram; Stop; %}
```

const_int,oper,ident	157
key,chr,oper	177
oper,const_int,oper	180
oper,oper,ident	181
ident,oper,chr	185
storage,type,ident	263
type,oper,ident	541
ident,oper,const_int	739
oper,ident,oper	2342
ident,oper,ident	4197

# cobra inline programs

## example 2

count the number of cases in  
a switch, taking into account that  
switch statements may be nested.

```
$ cobra -f nr_cases_all cobra_lib.c | sort -n
 3 cobra_lib.c:173
 3 cobra_lib.c:1993
 3 cobra_lib.c:538
 3 cobra_lib.c:683
 4 cobra_lib.c:3450
 7 cobra_lib.c:3416
 8 cobra_lib.c:1717
 8 cobra_lib.c:583
16 cobra_lib.c:1597
27 cobra_lib.c:1546
```

```
%{
  if (.curly > 0 && #switch)
  { q = .;
    . = .nxt;
    if (.txt != "(" )
    {      . = q;
          Next;
    }
    . = .forw;
    . = .nxt;
    if (.txt != "{" )
    {      . = q;
          Next;
    }
    q.mark = 0;
    while (.curly >= q.curly)
    {      if (.curly == q.curly + 1
            && (#case || #default))
            {      q.mark++;
            }
            . = .nxt;
    }
    print q.mark " " .fnm ":" q.lnr "\n";
    . = q;
  }
%}
```

# cobra inline programs

## example 3

```
%{  
  if (#float)  
  {  
    . = .nxt;  
    if (@ident)  
    {  
      Store[.txt] = .;      # store current location  
      print .fnm ":" .lnr ": declaration of '" .txt "'\n";  
    }  
    Next;  
  }  
  if (@ident)  
  {  
    q = Store[.txt];  
    if (q.seq != 0)  
    {  
      print .fnm ":" .lnr ": use of float '" .txt "' ";  
      print "declared at " q.fnm ":" q.lnr "\n";  
    }  
  }  
%}
```

# cobra inline programs

## example 4 – propagating data forward

```
%{      # check the identifier length for all tokens
        # and remember the longest in variable q

        if (@ident && .len > q.len)
        {           q = .;
        }

%}
# q holds its last value
%{
    print q.fnm ":" q.lnr ": " q.txt " = " q.len " chars\n";
    Stop; # stops after the line is printed
%}
```

```
cobra_links.c:487: switch_links_range = 18 chars
```

# cobra inline programs

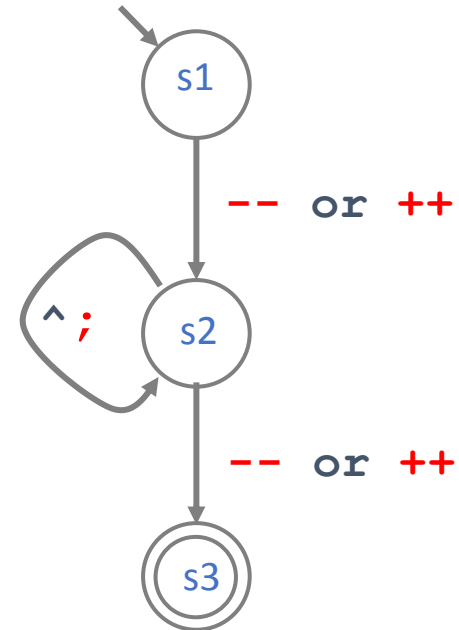
script for matching "[-- ++] ^;\* [-- ++]"

*in earlier versions of Cobra, the DFA was converted into an inline program (the current version uses the DFA directly)*

```
%{
    q = .;
s1:   if (q.txt == "--" || q.txt == "++")
      { q = q.nxt; goto s2;
      } else { Next; }

s2:   if (q.txt == ';' ) { Next; }
      if (q.txt == "-" || q.txt == "+")
      { q = q.nxt; goto s3;
      } else { q = q.nxt; goto s2; }

s3:   r = .;
      while (r != q) { r.mark++; r = r.nxt; }
      Next;
}%
```



# cobra inline programs

script for matching `"for ( x:@ident .* ) { .* :x .* }"`

```
%{
    q = .;
S0:
    if (q.txt == "for") {
        if (q == q.nxt) { Next; }
        q = q.nxt; goto S1;
    }
    Next;
S1:
    if (q.txt == "(") {
        if (q == q.nxt) { Next; }
        p_lft = q;
        q = q.nxt; goto S2;
    }
    Next;
S2:
    if (q.typ == "ident") {
        if (q == q.nxt) { Next; }
        x = q;
        q = q.nxt; goto S3;
    }
    Next;
S3:
    if (q.txt == ")") {
        if (q == q.nxt) { Next; }
        if (q.round != p_lft.round)
        { q = q.nxt; goto S3; }
        q = q.nxt; goto S4;
    }
    if (q == q.nxt) { Next; }
    q = q.nxt; goto S3;
}
```

*name binding*



```
S4:
    if (q.txt == "{") {
        if (q == q.nxt) { Next; }
        c_lft = q;
        q = q.nxt; goto S5;
    }
    Next;
S5:
    if (q.txt == x.txt) {
        if (q == q.nxt) { Next; }
        q = q.nxt; goto S6;
    }
    if (q == q.nxt) { Next; }
    q = q.nxt; goto S5;
S6:
    if (q.txt == ")") {
        if (q == q.nxt) { Next; }
        if (q.curly != c_lft.curly) { q = q.nxt; goto S6; }
        q = q.nxt; goto S7;
    }
    if (q == q.nxt) { Next; }
    q = q.nxt; goto S6;
S7:
    r = .;
    while (r != q)
    {
        r.mark = 1;
        r = r.nxt;
    }
    Next;
}%
```

*reference to the bound name x*



# cobra inline programs

## recursive functions

```
$ cobra some_file.c
: %{
    function fact(n)
    {   if (n <= 1)
        {   return 1;
            }
        return n*fact(n-1);
    }

    print "10! = " fact(10) "\n";
    Stop;
%}
10! = 3628800
:
```

remember:

inline programs are  
by default executed  
once for every token  
in the input stream

this has two consequences:

1. there has to be minimally one token to process
2. if something is meant to be executed only once, we need to explicitly Stop the control loop

# cobra inline programs

creating new tokens

```
%{  
    a = newtok();      # create 3 new empty tokens  
    b = newtok();  
    c = newtok();  
  
    a.txt = "2";      # assign the .txt fields  
    b.txt = "+";  
    c.txt = "2";  
  
    a.typ = "oper";   # assign the .typ field  
  
    a.nxt = b;        # connect a to b  
    b.nxt = c;        # and b to c  
    set_ranges(a, c); # define a range from a to c  
    Stop;  
%}  
%{  
    print .txt "\n";  # scan the newly defined range  
%}
```

running this program on  
arbitrary input, prints:

2  
+  
2

# cobra inline programs

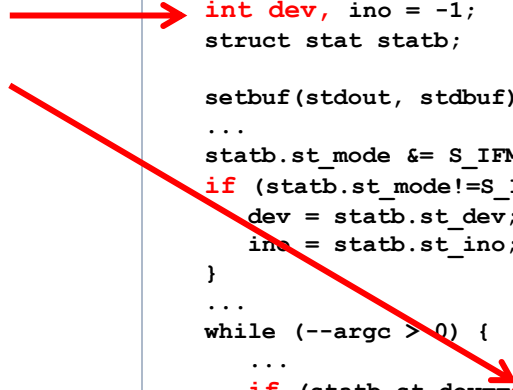
dealing with flow-sensitive properties: uninitialized variable use

```
$ cd Unix/V7/usr/src/cmd  
$ cobra -f play/dfs_uninit *.c
```

```
...  
cat.c:16 declaration of dev  
cat.c:50 possible uninitialized use
```

*dfs\_uninit* adds links to capture a basic control-flow graph information for each function (if/else/goto) and then uses recursive calls to perform a depth-first search over the control-flow graphs to find the suspicious execution paths

```
main(argc, argv)  
char **argv;  
{  
  ...  
  int dev, ino = -1;  
  struct stat statb;  
  
  setbuf(stdout, stdbuf);  
  ...  
  statb.st_mode &= S_IFMT;  
  if (statb.st_mode!=S_IFCHR && statb.st_mode!=S_IFBLK) {  
    dev = statb.st_dev;  
    ino = statb.st_ino;  
  }  
  ...  
  while (--argc > 0) {  
    ...  
    if (statb.st_dev==dev && statb.st_ino==ino) {  
      fprintf(stderr, "cat: input %s is output\n",  
              fflg?"-": *argv);  
      fclose(fi);  
      continue;  
    }  
    ...  
  }  
}
```



159 .c files, 30 KLOC, 1 core, 0.8 seconds  
5 accurate warnings + 1 false positive

# cobra inline programs

dealing with flow-sensitive properties

[play/goto\\_links.cobra](#)

collects goto statements and labels  
and connects the `.bound` field of gotos to  
the corresponding target label

similarly [break\\_links.cobra](#), [else\\_links.cobra](#),  
and [switch\\_links.cobra](#) use the `.bound` token  
attribute to set shortcuts, e.g. from `case` label  
to `case` label, or from `if` to `else`, etc.

a relevant fragment from `else_links.cobra`:

```
if (.txt == "if")
{
    q = .;
    skip_cond();
    while (.typ == "cmnt")
    {
        . = .nxt;
    }
    if (.txt == "{")
    {
        . = .jmp;
        . = .nxt;
    } else
    {
        skip_stmnt();
    }
    if (.txt == "else")
    {
        q.bound = .nxt;
    } else
    {
        q.bound = .;
    }
    . = q;
    Next;
}
```

# cobra inline programs

using multiple cores – extending the earlier Trigram program

```
: ncore 8          # use 8 cores, or: cobra -N8 *.c
```

```
%{
    q = .nxt;
    r = q.nxt;
    if (.typ != "" && q.typ != "" && r.typ != "")
    {
        Trigram[.typ, q.typ, r.typ]++;
    }
}%
track start _tmp_
%{
    if (cpu != 0)
    {
        Stop;
    }
    a_unify(0);
    for (i in Trigram)
    {
        print i.txt "\t" sum(Trigram[i.txt]) "\n";
    }
    Stop;
}%
track stop
!sort -k2 -n < _tmp_ | tail -10; rm -f _tmp_
```

post-process on cpu 0 only

unify the associative array data  
from all cores and make it available  
to cpu 0

sum the collected results

# cobra inline programs

## concurrency control

```
$ cobra -N4 *.c # cobra sources
4 cores, 10 files, 56546 tokens
%{
  if (.txt == "for") # for is a keyword, so #for doesn't work here
  { count++; # { ... } braces always required
  }
%}
%{
  lock();
  print cpu ": my count = " count "\n";
  unlock();
  if (cpu == 0)
  { print cpu ": total = " sum(count) "\n"; # only 1 cpu gets here
  }
  Stop;
%}
```

when multiple cores are used,  
each core scans part of the input  
sequence, so Begin and End refer  
to the local part  
when needed, the very first and  
very last token can be accessed  
via `first_t` and `last_t`

```
0: my count = 54
0: total = 210
1: my count = 50
2: my count = 55
3: my count = 51
```

# cobra inline programs

## concurrency control

when multiple cores are used,  
each core scans part of the input  
sequence, so **Begin** and **End** refer  
to the local part

when needed, the very first and  
very last token can be accessed  
via **first\_t** and **last\_t**

```
# let cpu 0 scan all tokens backwards
# for no good reason....
$ cobra -N4 *.c
4 cores, 10 files, 56546 tokens
%{
  if (cpu == 0)
  {   . = last_t;           # start at the end
      while (. != first_t) # to the beginning
      {   .mark = .seq;     # something pointless
          . = .prv;        # backwards
      }
  }
  Stop;
%}
```

building standalone checkers

# standalone checkers

linked to the cobra front-end

we can write standalone checkers using the infrastructure that is built by Cobra used as a front-end, to get the full power of C.

the structure of a standalone checker is defined as follows:

```
1  #include "c_api.h"
2
3  typedef struct Names Names;
4  struct Names {
5      char *nm;
6      int cnt;
7      Names *nxt;
8  } *names;
9
10 int
11 newname(char *s)
12 {
13     ...
14
15     ...
16 }
17
```

example checkers of this type are included in the distribution in the `src_app` subdirectory, including checkers for a range of `cwe` properties defined multi-threaded

```
28 void
29 cobra_main(void)           // the interface point
30 {
31     for (cur = prim; cur; NEXT) // main loop over the token sequence
32     { if (TYPE("ident"))       // if (strcmp(cur->typ, "ident") == 0)
33         { if (verbose)
34             { printf("n_%d ", newname(cobra_txt()));
35             } else
36             { printf("ident ");
37             }
38         } else
39         { printf("%s ", cur.txt);
40         }
41         if (MATCH(";")        // if (strcmp(cur->txt, ";") == 0)
42             || MATCH("}")
43             || TYPE("cpp"))
44             { printf("\n");
45             }
46         }
47     }
```

# standalone checkers

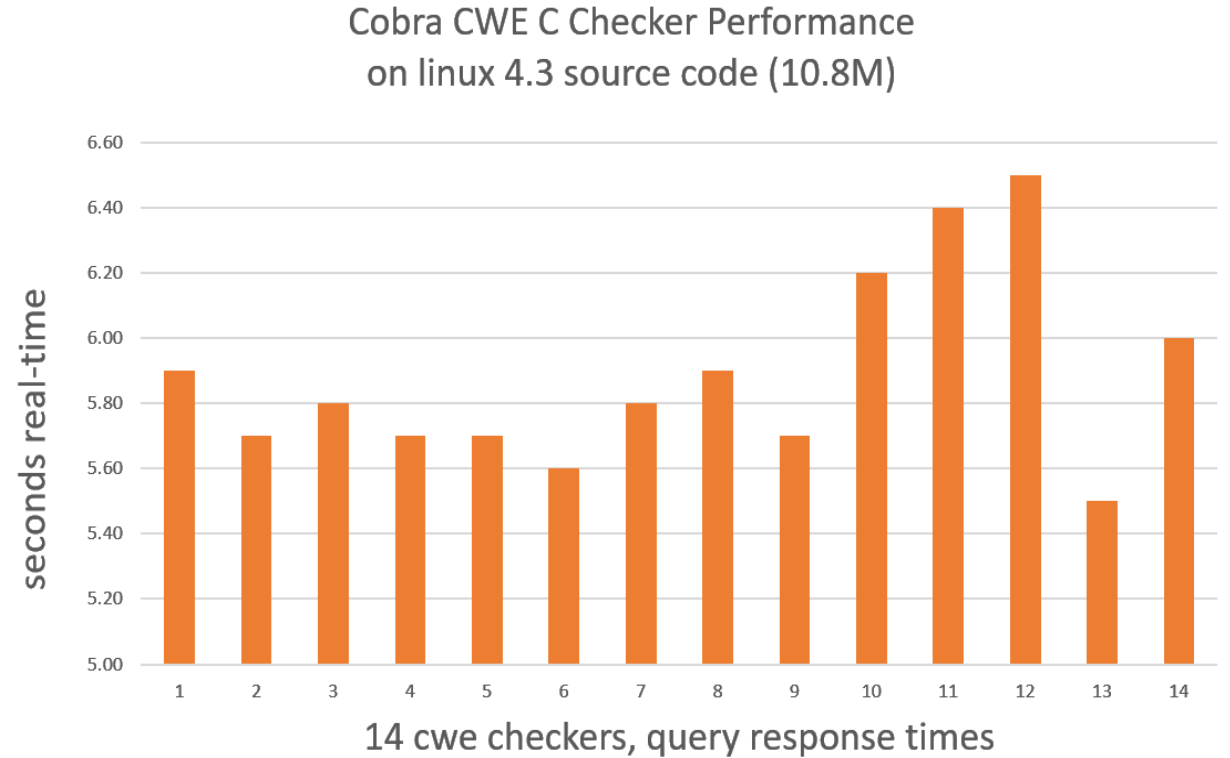
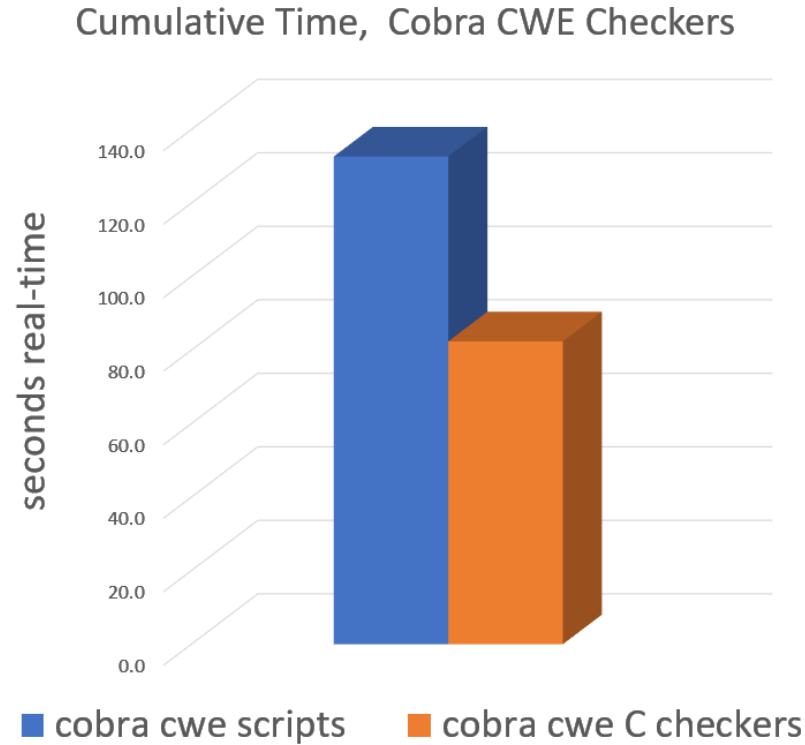
the multi-threaded cwe checkers in `$(COBRA)/src_app`  
and the precompiled binary in `$(COBRA)/bin_...`

```
$ ls -l src_app/cwe*
-rw-r--r--+ 1 gh None 2587 Nov 29 2018 cwe.c
-rw-r--r--+ 1 gh None 1018 May 6 13:39 cwe.h
-rw-r--r--+ 1 gh None 12089 Apr 26 13:10 cwe_119.c
-rw-r--r--+ 1 gh None 9236 Apr 27 09:20 cwe_120.c
-rw-r--r--+ 1 gh None 5291 Mar 14 13:44 cwe_131.c
-rw-r--r--+ 1 gh None 3450 Apr 27 10:26 cwe_134.c
-rw-r--r--+ 1 gh None 4105 Mar 14 13:44 cwe_170.c
-rw-r--r--+ 1 gh None 6435 Mar 14 13:54 cwe_197.c
-rw-r--r--+ 1 gh None 8216 May 6 13:30 cwe_416.c
-rw-r--r--+ 1 gh None 10934 May 6 13:30 cwe_457.c
-rw-r--r--+ 1 gh None 1467 Mar 8 14:12 cwe_468.c
-rw-r--r--+ 1 gh None 4423 Mar 8 14:53 cwe_805.c
-rw-r--r--+ 1 gh None 6335 May 6 13:35 cwe_util.c
$ ls -l bin_cygwin/cwe*
-rwxr-xr-x+ 1 USER None 271152 Jun 4 09:21 ../bin_cygwin/cwe.exe
```

for comparison:  
the cobra scripted equivalents  
for each cwe check are also available  
in `$(COBRA)/rules/cwe/...`

# standalone checkers

performance, compared with scripted checkers on 18.6 MLOC of source code (linux 4.3)



C standalone: response times: 5.5 – 6.5 seconds per CWE check (single core)

Cobra scripted: 1.6x slower

Startup time: ~10 seconds multi-core

*thank you!*

manual pages, tutorials, papers:

<http://www.spinroot.com/cobra>

source code, rule libraries, binaries:

<https://github.com/nimble-code/Cobra>

