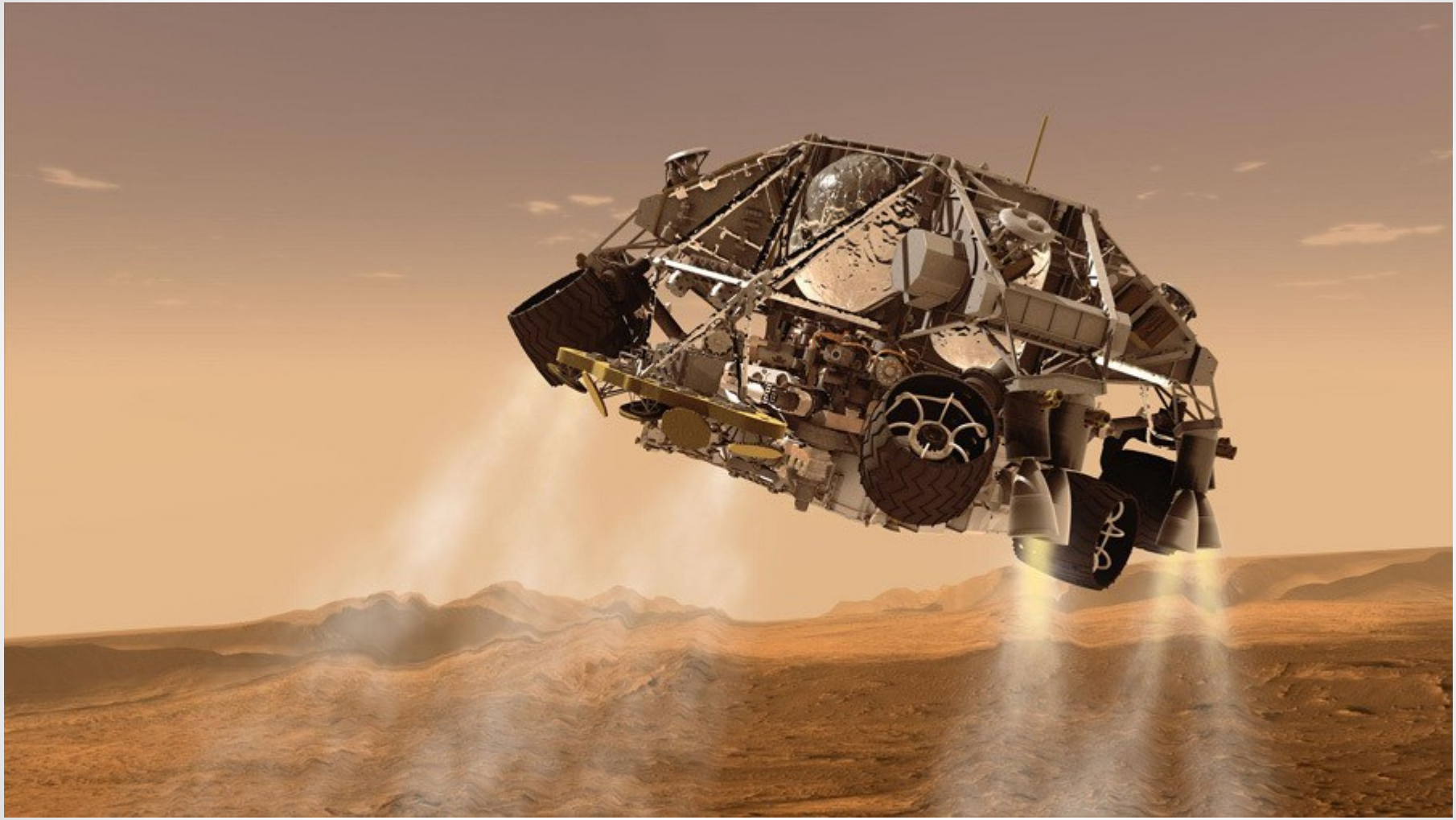
Cobra:
Fast Structural Code Checking

Gerard J. Holzmann
Nimble Research
SPIN 2017

when it really *has* to work…

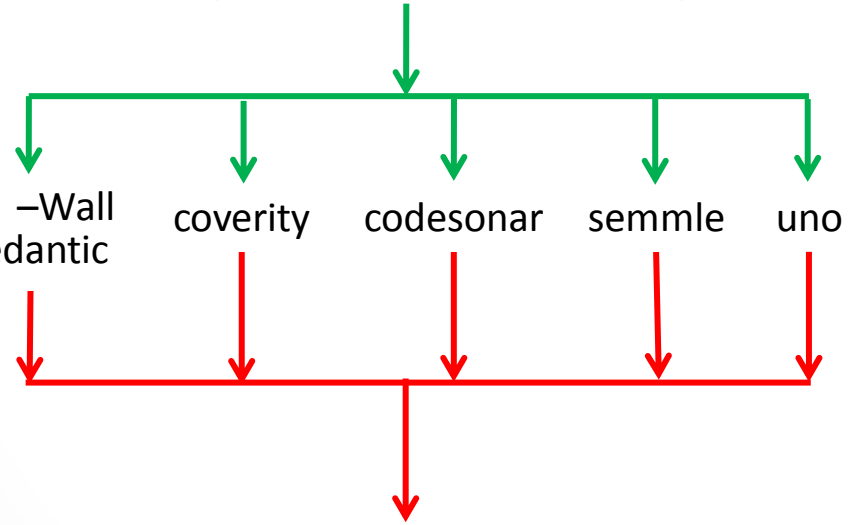


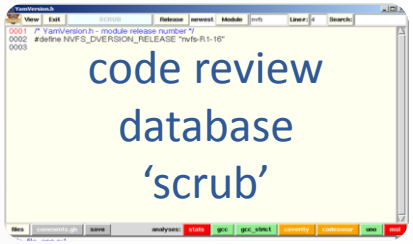
tool-based code review @ jpl

~2.8 M lines of C

Nightly Build Log
~3K compiler calls

Static Code Analysis for
Defect Detection &
Coding Rule Compliance Checking



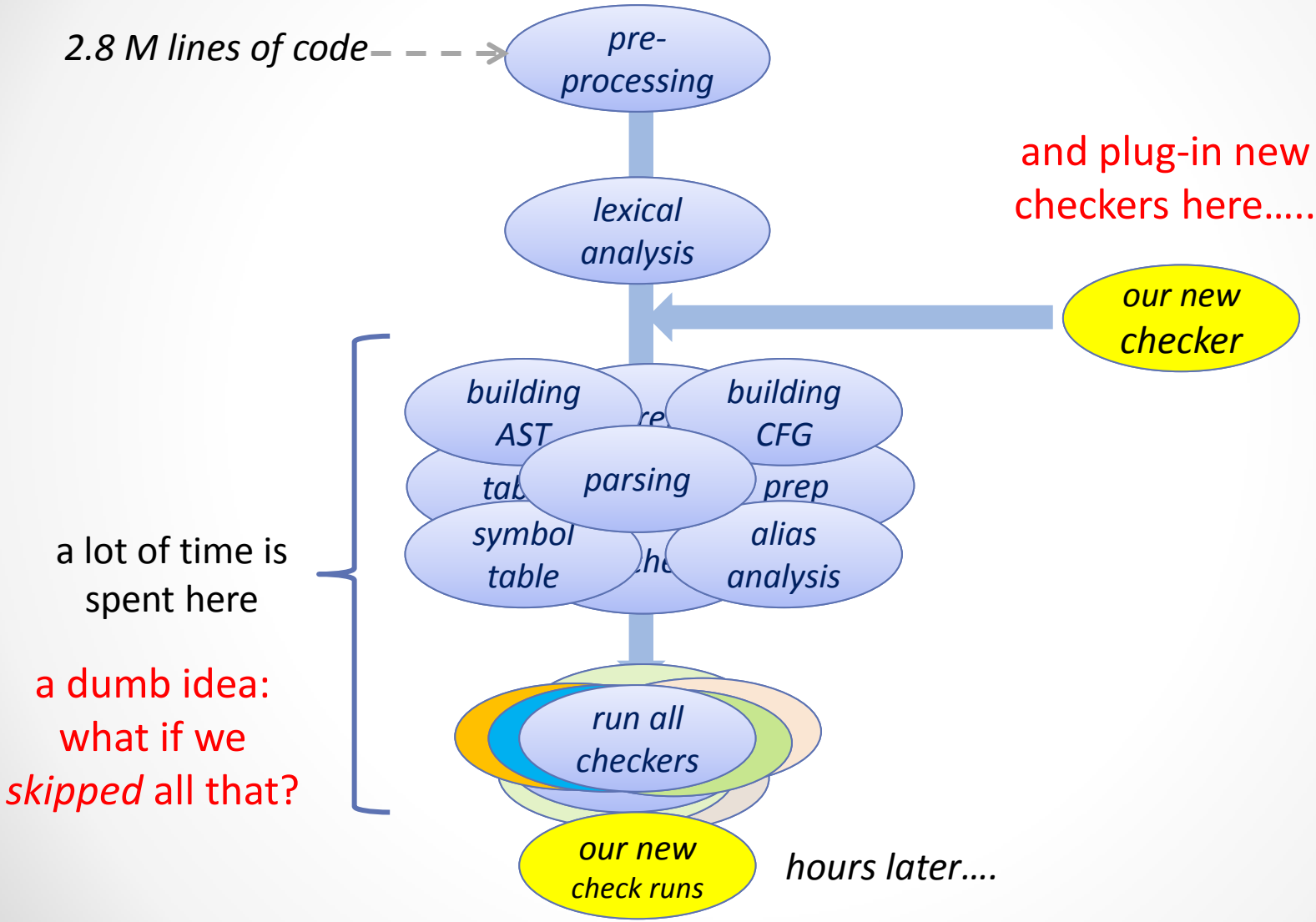


total
analysis time
~15hrs

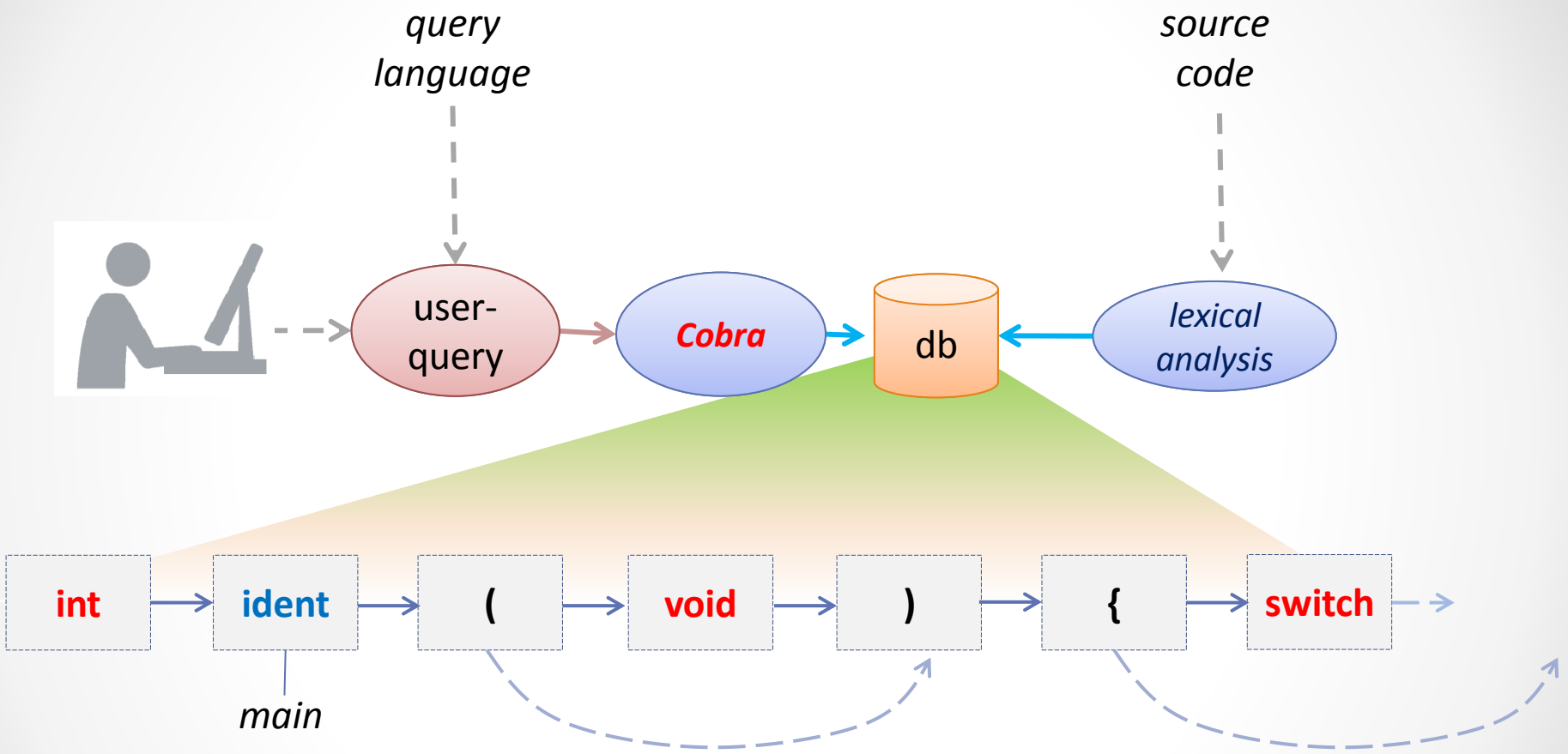
now consider this scenario

- an in-flight anomaly occurs
  - manual analysis reveals the cause:
  - a function 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
- *Q: can this happen anywhere else in the code?*
  - write new checkers for some of the static analyzers
  - wait 15 hours for the cumulative check to be completed…
  - meanwhile, on a few million miles away…..

so, why is the analysis taking so long?



cobra



a linked list of lexical tokens with annotations

(token types, ranges, levels of nesting for parentheses, brackets, and braces, etc.)

now we can do pattern matching on tokens

- first the obvious:

```
$ cobra -cpp -e j *.[ch]
 # compare with: grep -e j *.[ch]
 # differences:
 #     we match tokens, not characters
 #     code is optionally pre-processed
```

- match on token types:

```
$ cobra -e @qualifier *.[ch]
 # e.g., const, volatile
$ cobra -e @modifier *.[ch]
 # e.g., long, short, unsigned
```

- matching regular expressions:

```
$ cobra -e '{ .* malloc ^free }' *.[ch]
 # functions calling malloc, but not free
```

using sets and ranges

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

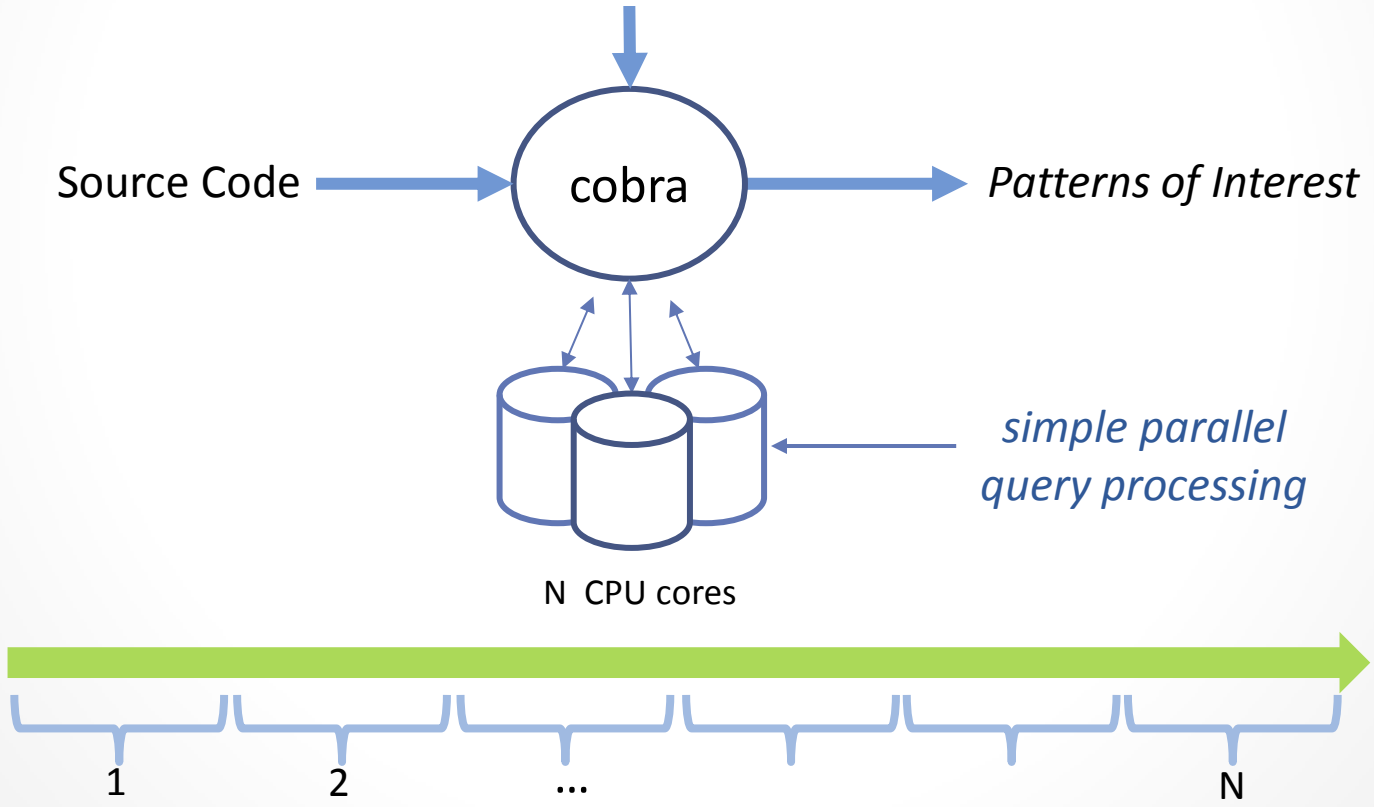
```
478             switch (f->n->ntyp) {
479             case UNLESS:
480                     attach_escape(f->sub->this, e);
481                     break;
482             case IF:
483             case DO:
484                     for (z = f->sub; z; z = z->nxt)
485                             attach_escape(z->this, e);
486                     break;
487             case D_STEP:
488                     /* attach only to the guard stmnt */
489                     escape_el(f->n->sl->this->frst, e);
490                     break;
491             case ATOMIC:
492             case NON_ATOMIC:
493                     /* attach to all stmnts */
494                     attach_escape(f->n->sl->this, e);
495                     break;
> 496             }
...
user    0m0.288s
sys     0m0.032s
$
```

example query:
match *switch* statements
without a *default* clause

m[ark] defines a set
n[ext] moves the match position forward
{ … } is a predefined range that we can query

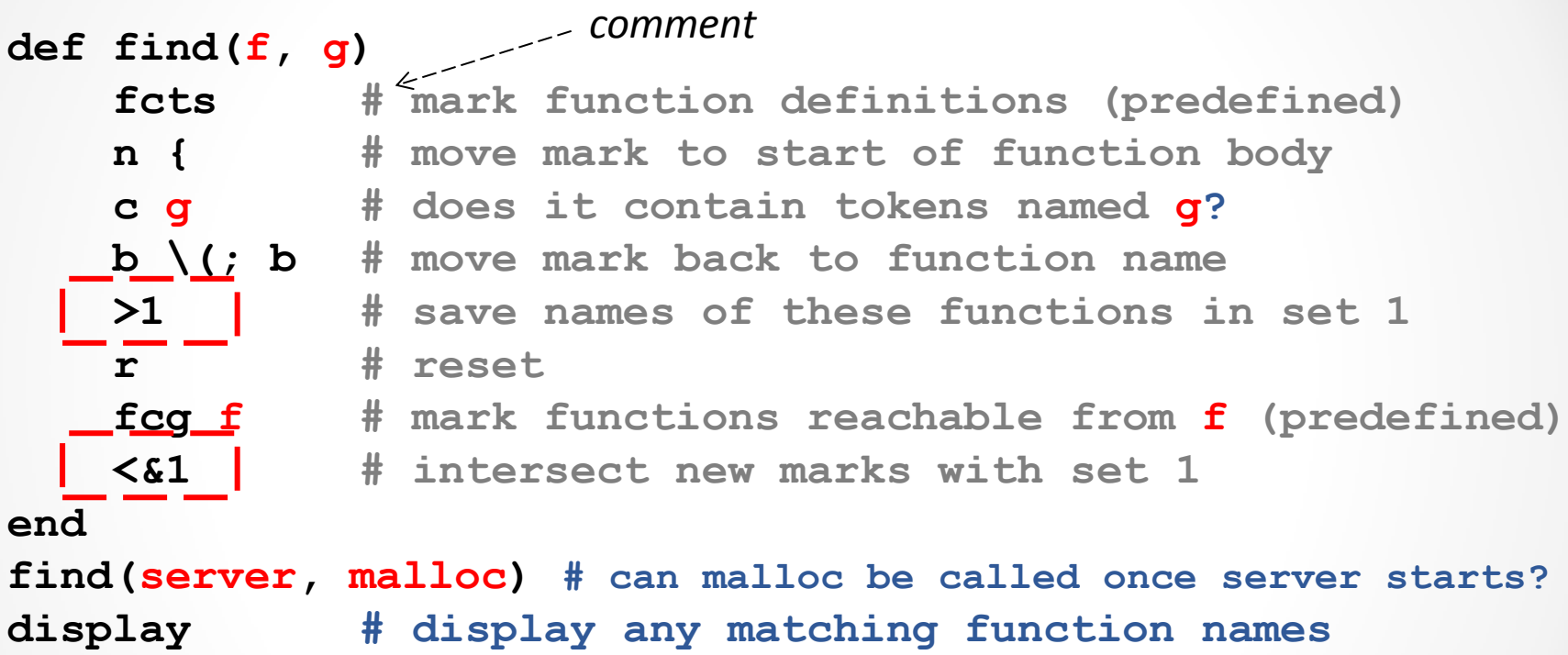
context: querying code

- interactive queries (sets, ranges)
- regular expression pattern matching
- inline programs



command scripts and creating sets

example: are function calls to g reachable from f?



predefined operations on sets

```
>n  # save       current marks in set n
<n  # assign:    replace current marks with set n
<|n # union:     add marks from set n to current
<&n # intersect: keep only marks also in set n
<^n # subtract:  keep only marks not in set n
```

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 all operations)

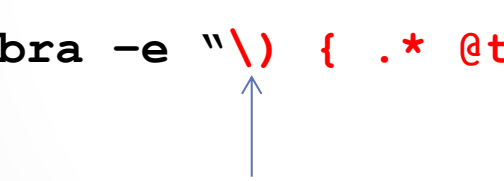
variable binding

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

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

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

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



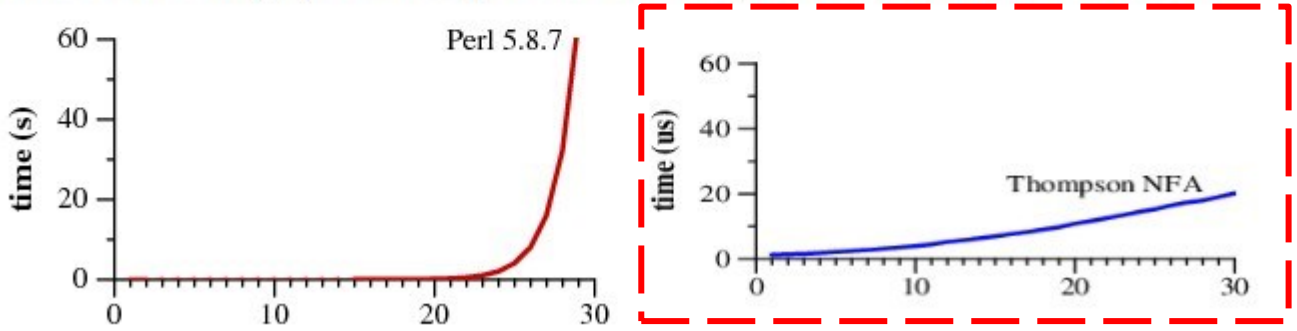
implementation of the matching algorithm

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

Russ Cox
rsc@swtch.com
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

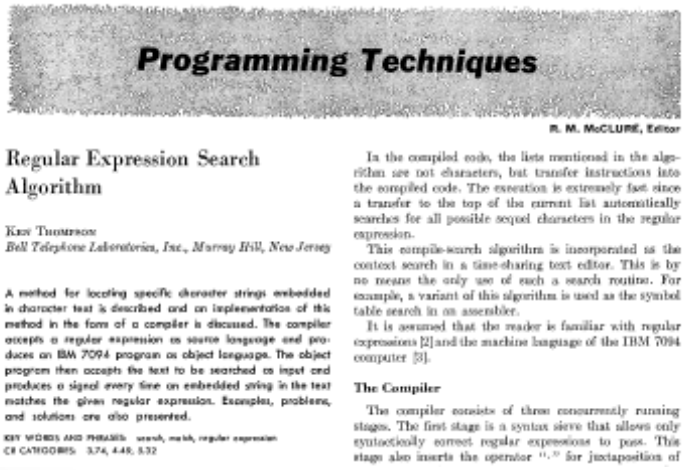
ken thompson's algorithm

example: find expressions with multiple side-effects

$ cobra -e '(--|\+\+) ^;* (--|\+\+)' *.c

sml_dsa.c:

```
17: 213     lA1 = (ulong) (*p++)*(*q--);
```



Programming Techniques

R. M. McCLURE, Editor

Regular Expression Search Algorithm

Ken Thompson
Bell Telephone Laboratories, Inc., Murray Hill, New Jersey

A method for locating specific character strings embedded in character text is described and an implementation of this method in the form of a compiler is discussed. The compiler accepts a regular expression as source language and produces an IBM 7094 program as object language. The object program then accepts the text to be searched as input and produces a signal every time an embedded string in the text matches the given regular expression. Examples, problems, and solutions are also presented.

KEY WORDS AND PHRASES: search, match, regular expression
CR CATEGORIES: 3.74, 4.49, 5.32

In the compiled code, the lists mentioned in the algorithm are not characters, but transfer instructions into the compiled code. The execution is extremely fast since a transfer to the top of the current list automatically searches for all possible sequel characters in the regular expression.

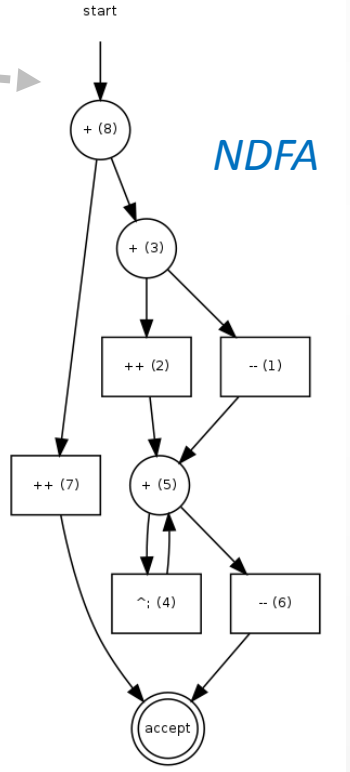
This compile-search algorithm is incorporated in the context search in a time-sharing text editor. This is by no means the only use of such a search routine. For example, a variant of this algorithm is used in the symbol table search in an assembler.

It is assumed that the reader is familiar with regular expressions [2] and the machine language of the IBM 7094 computer [3].

The Compiler

The compiler consists of three concurrently running stages. The first stage is a syntax sieve that allows only syntactically correct regular expressions to pass. This stage also inserts the operator "." for juxtaposition of

Thompson's algorithm
(CACM 11:6 1968)



NDFA

cobra –view –e '…'

inline Cobra programs

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

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

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

unless we Stop the execution explicitly

example: a recursive function

writing inline Cobra programs

- inline programs can access (and modify) all token attributes, use associative arrays, recursive functions,

```
strings:
        .fct            # function name
        .fnm            # file name
        .txt            # token text
        .typ            # token type
        .fct            # name of containing function, or "global"

numbers:
        .round          # nesting level of ()
        .bracket        # nesting level of []
        .curly          # nesting level of {}
        .len            # length of token text
        .lnr            # linenumber
        .mark           # user-definable integer value
        .seq            # token sequence number
        .range          # the nr lines in the associated range

tokens:
        .nxt            # the immediately following token
        .prv            # the immediately preceding token
        .jmp            # move to other end of range, eg from { to } or back
        .bound          # link to bound symbol reference
```

quite simple inline programs can already do useful things

```
%{ # check the identifier length for all tokens
   # and remember the longest in a variable q
   if (@ident && .len > q.len){ q = .; }
%}

   # the scan over all tokens is now completed,
   # q retains its value, which can now be printed:

%{
  print "longest ident is: " q.txt " = " q.len " chars\n";
  Stop; # stops after the above line is printed
%}
```

associative arrays & external commands

e.g., find the most common token-type trigrams

```
%{
        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_
```

associative array

divert output to a temporary file

post-process with cpu 0 only

collect data from all cores at cpu 0

shell escape

the 10 most frequently occurring type trigrams

```
$ cobra -f play/trigram *.[ch]
ident,oper,chr              209
const_int,oper,ident        231
oper,oper,ident             232
storage,type,oper           239
key,const_int,oper          250
storage,type,ident          702
ident,oper,const_int        1000
type,oper,ident             1298
oper,ident,oper             3541
ident,oper,ident            5695
```

finding uninitialized variable use (a flow-sensitive property)

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

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

(1) the script creates links to capture a rudimentary control-flow graph for each function (if/else/goto)

(2) using recursive fct calls, it then performs a DFS over the CFG to find 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

so does it scale?
multi-core processing

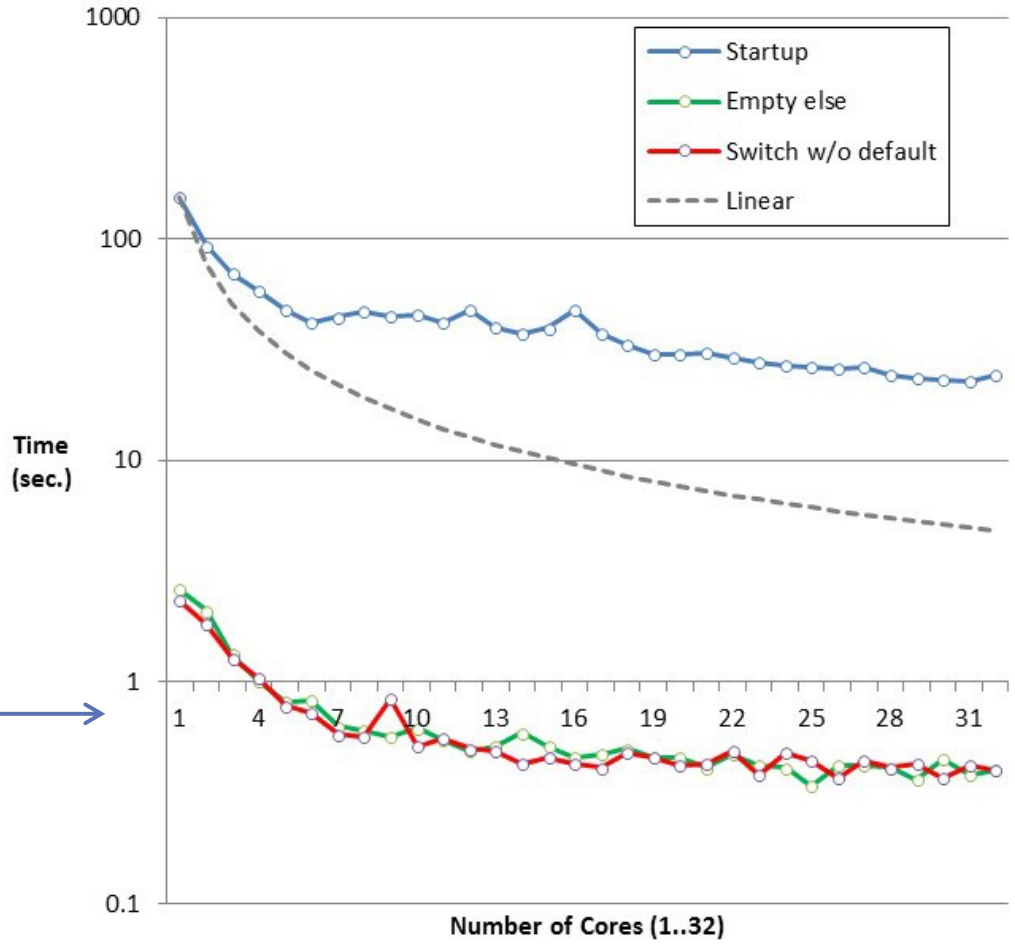
18,633,817 Lines of Code of the Linux 4.3 distribution, with 39,144 .c and .h files

checking 2 types of queries:

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

using 1..32 CPU cores

4 cores or more:
query processing < 1 sec.



documentation, manuals, downloads:
www.spinroot.com/cobra

thanks!