Regular Expression Matching: the Virtual Machine Approach

Russ Cox
rsc@swtch.com
December 2009

Introduction

Name the most widely used bytecode interpreter or virtual machine. Sun's JVM? Adobe's Flash? .NET and Mono? Perl? Python? PHP? These are all certainly popular, but there's one more widely used than all those combined. That bytecode interpreter is Henry Spencer's regular expression library and its many descendants.

The first article in this series described the two main strategies for implementing regular expression matching: the worst-case linear-time NFA- and DFA-based strategies used in awk and egrep (and now most greps), and the worst-case exponential-time backtracking strategy used almost everywhere else, including ed, sed, Perl, PCRE, and Python.

This article presents two strategies as two different ways to implement a virtual machine that executes a regular expression that has been compiled into text-matching bytecodes, just like .NET and Mono are different ways to implement a virtual machine that executes a program that has been compiled into CLI bytecodes.

Viewing regular expression matching as executing a special machine makes it possible to add new features just by adding (and implementing!) new machine instructions. In particular, we can add regular expression submatching instructions, so that after matching `(a+)(b+)` against `aabbbb`, a program can find out that the parenthesized `(a+)` (often referred to as `\1` or `$1`) matched `aa` and that `(b+)` matched `bbbb`. Submatching can be implemented in *both* backtracking and non-backtracking VMs. (Code doing this dates back to 1985, but I believe this article is the first written explanation of it.)

A Regular Expression Virtual Machine

To start, we'll define a regular expression virtual machine (think Java VM). The VM executes one or more threads, each running a regular expression program, which is just a list of regular expression instructions. Each thread maintains two registers while it runs: a program counter (PC) and a string pointer (SP).

The regular expression instructions are:

`char` c	If the character SP points at is not c, stop this thread: it failed. Otherwise, advance SP to the next character and advance PC to the next instruction.
`match`	Stop this thread: it found a match.
`jmp` x	Jump to (set the PC to point at) the instruction at x.
`split` x, y	Split execution: continue at both x and y. Create a new thread with SP copied from the current thread. One thread continues with PC x. The other continues with PC y. (Like a simultaneous jump to *both* locations.)

The VM starts with a single thread running with its PC pointing at the beginning of the program and its SP pointing at the beginning of the input string. To run a thread, the VM executes the instruction that the thread's PC points at; executing the instruction changes the thread's PC to point at the next instruction to run. Repeat until an instruction (a failed `char` or a `match`) stops the thread. The regular expression matches a string if any thread finds a match.

Compiling a regular expression into byte code proceeds recursively depending on the form of the regular expression. Recall from the previous article that regular expressions come in four forms: a single letter like `a`, a concatenation e_1e_2, an alternation $e_1|e_2$, or a repetition e? (zero or one), e* (zero or more), or e+ (one or more).

A single letter `a` compiles into the single instruction `char a`. A concatenation concatenates the compiled form of the two subexpressions. An alternation uses a `split` to allow either choice to succeed. A zero-or-one repetition e? uses a `split` to compile like an alternation with the empty string. The zero-or-more repetition e* and the one-or-more repetition e+ use a `split` to choose whether to match e or break out of the repetition.

The exact code sequences are:

`a`		`char a`
e_1e_2		*codes for* e_1
		codes for e_2
$e_1|e_2$		`split L1, L2`
	`L1:`	*codes for* e_1
		`jmp L3`

```
                L2: codes for e2
                L3:
e?                  split L1, L2
                L1: codes for e
                L2:
e*              L1: split L2, L3
                L2: codes for e
                    jmp L1
                L3:
e+              L1: codes for e
                    split L1, L3
                L3:
```

Once the entire regular expression has been compiled, the generated code is finished with a final `match` instruction.

As an example, the regular expression `a+b+` compiles into

```
0   char a
1   split 0, 2
2   char b
3   split 2, 4
4   match
```

When run on `aab`, a VM implementation might run the program this way:

Thread	PC	SP	Execution
T1	`0 char a`	**a**ab	character matches
T1	`1 split 0, 2`	a**a**b	creates thread T2 at PC=2 SP=a**a**b
T1	`0 char a`	a**a**b	character matches
T1	`1 split 0, 2`	a**a**b	creates thread T3 at PC=2 SP=aa**b**
T1	`0 char a`	aa**b**	no match: thread T1 dies
T2	`2 char b`	a**a**b	no match: thread T2 dies
T3	`2 char b`	aa**b**	character matches
T3	`3 split 2, 4`	abb_	creates thread T4 at PC=4 SP=abb_
T3	`2 char b`	abb_	no match (end of string): thread T3 dies
T4	`4 match`	abb_	match!

In this example, the implementation waits to run a new thread until the current thread finishes, and it runs the threads in the order they were created (oldest first). This is not required by the VM specification; it is up to the implementation to schedule the threads. Other implementations might run the threads in another order or might even interleave thread executions.

VM Interface in C

The rest of this article examines a sequence of VM implementations, illustrating them using C source code. The regular expression program is represented as an array of `Inst` structures, defined in C as:

```
enum {    /* Inst.opcode */
    Char,
    Match,
    Jmp,
    Split
};

struct Inst {
    int opcode;
    int c;
    Inst *x;
    Inst *y;
};
```

This bytecode is almost identical to the representation of the NFA graphs from the first article. We can view the bytecode as an encoding of the NFA graphs into machine instructions, or we can view the NFA graphs as the control-flow graph of the bytecodes. Each view makes different things easier to think about. Whichever came first, this article focuses on the of the machine instruction view.

Each VM implementation will be a function taking the program and input string as arguments and returning an integer signaling whether the program matched the input string (zero for no match; non-zero for a match).

```
int implementation(Inst *prog, char *input);
```

A Recursive Backtracking Implementation

The simplest possible implementation of the VM doesn't model the threads directly at all. Instead, it calls itself recursively when it needs to explore a new thread of execution, taking advantage of the fact that the `prog` and `input` function parameters double as the initial values of `pc` and `sp` for the first thread.

```
int
recursive(Inst *pc, char *sp)
{
    switch(pc->opcode){
    case Char:
        if(*sp != pc->c)
            return 0;
        return recursive(pc+1, sp+1);
    case Match:
        return 1;
    case Jmp:
        return recursive(pc->x, sp);
    case Split:
        if(recursive(pc->x, sp))
            return 1;
        return recursive(pc->y, sp);
    }
    assert(0);
    return -1;  /* not reached */
}
```

The above version is very recursive and should be comfortable to programmers familiar with recursion-heavy languages like Lisp, ML, and Erlang. Most C compilers will rewrite the "`return recursive(...);`" statements (the so-called tail calls) into a goto back to the top of the function, so that the above compiles into something more like:

```
    int
    recursiveloop(Inst *pc, char *sp)
    {
        for(;;){
            switch(pc->opcode){
            case Char:
                if(*sp != pc->c)
                    return 0;
                pc++;
                sp++;
                continue;
            case Match:
                return 1;
            case Jmp:
                pc = pc->x;
                continue;
            case Split:
                if(recursiveloop(pc->x, sp))
                    return 1;
                pc = pc->y;
                continue;
            }
            assert(0);
            return -1;  /* not reached */
        }
    }
```

where the looping is explicit.

Note that this version still needs one (non-tail) recursion, in `case Split`, to try `pc->x` before trying `pc->y`.

This implementation is the essence of Henry Spencer's original library as well as the backtracking implementations in Java, Perl, PCRE, Python, and the original ed, sed, and grep. This implementation runs very fast when there is little backtracking to be done, but it slows considerably when many possibilities must be tried, as we saw in the previous article.

This particular backtracking implementation has one shortcoming that production implementations usually do not: regular expressions like `(a*)*` can cause infinite loops in the compiled program, and this VM implementation does not detect such loops. It turns out to be easy to fix this problem (see the end of the article for details), but since backtracking is not our focus, we will simply ignore the problem.

A Non-recursive Backtracking Implementation

The recursive backtracking implementation runs a single thread until it dies, and then runs threads in the reverse of the order in which they were created (newest first). The threads waiting to run are not encoded explicitly: instead they are implicit in the values of `pc` and `sp` saved on the C call stack whenever the code recurses. If there are too many threads waiting to run, the C call stack can overflow, causing errors much harder to debug than a performance problem. The problem most commonly comes up during repetitions like `.*`, which create a new thread after each character (like `a+` did in the example above). This is a real concern in multithreaded programs, which often have limited stack sizes and no hardware checking for stack overflow.

We can avoid overflowing the C stack by maintaining an explicit thread stack instead. To start, we define a `struct Thread` and a simple constructor function `thread`:

```
struct Thread {
    Inst *pc;
    char *sp;
};

Thread thread(Inst *pc, char *sp);
```

Then the VM implementation repeatedly takes a thread off its `ready` list and runs it to completion. If one thread finds a match, we can stop early: the remaining threads need not be run. If all threads finish without finding a match, there is no match. We impose a simple limit on the number of threads waiting to run, reporting an error if the limit is reached.

```
int
backtrackingvm(Inst *prog, char *input)
{
    enum { MAXTHREAD = 1000 };
    Thread ready[MAXTHREAD];
    int nready;
    Inst *pc;
    char *sp;

    /* queue initial thread */
    ready[0] = thread(prog, input);
    nready = 1;
```

```
        /* run threads in stack order */
        while(nready > 0){
            --nready;  /* pop state for next thread to run */
            pc = ready[nready].pc;
            sp = ready[nready].sp;
            for(;;){
                switch(pc->opcode){
                case Char:
                    if(*sp != pc->c)
                        goto Dead;
                    pc++;
                    sp++;
                    continue;
                case Match:
                    return 1;
                case Jmp:
                    pc = pc->x;
                    continue;
                case Split:
                    if(nready >= MAXTHREAD){
                        fprintf(stderr, "regexp overflow");
                        return -1;
                    }
                    /* queue new thread */
                    ready[nready++] = thread(pc->y, sp);
                    pc = pc->x;  /* continue current thread */
                    continue;
                }
            }
        Dead:;
        }
        return 0;
    }
```

This implementation behaves the same as `recursive` and `recursiveloop`; it just doesn't use the C stack. Compare the two `Split` cases:

```
/* recursiveloop */             /* backtrackingvm */
case Split:                     case Split:
    if(recursiveloop(pc->x, sp))    if(nready >= MAXTHREAD){
        return 1;                       fprintf(stderr, "regexp overflow");
    pc = pc->y;                         return -1;
    continue;                       }
                                    /* queue new thread */
                                    ready[nready++] = thread(pc->y, sp);
                                    pc = pc->x;  /* continue current thread */
                                    continue;
```

The backtracking is still present, but the `backtrackingvm` code must do explicitly what the `recursiveloop` did implicitly: save the PC and SP that will be used after the recursion so that they can be tried if the current thread fails. Being explicit makes it possible to add the overflow check.

Thompson's Implementation

Viewing regular expression matching as running threads in a VM, we can present an alternate formulation of Ken Thompson's algorithm, one that is closer to Thompson's PDP-11 machine code than what was presented in the first article.

Thompson observed that backtracking required scanning some parts of the input string multiple times. To avoid this, he built a VM implementation that ran all the threads in lock step: they all process the first character in the string, then they all process the second, and so on. This is possible because newly created VM threads never look backward in the string, so they can be coerced into lock step with the existing threads.

Because all threads execute in lock step, they all have the same value for the string pointer, so it is no longer necessary to save as part of the thread state:

```
struct Thread
{
        Inst *pc;
};
Thread thread(Inst *pc);
```

In our framework, Thompson's VM implementation is:

```
int
thompsonvm(Inst *prog, char *input)
{
    int len;
    ThreadList *clist, *nlist;
    Inst *pc;
    char *sp;

    len = proglen(prog);  /* # of instructions */
    clist = threadlist(len);
    nlist = threadlist(len);

    addthread(clist, thread(prog));
    for(sp=input; *sp; sp++){
        for(i=0; i<clist.n; i++){
            pc = clist.t[i].pc;
            switch(pc->opcode){
            case Char:
                if(*sp != pc->c)
                    break;
                addthread(nlist, thread(pc+1));
                break;
            case Match:
                return 1;
            case Jmp:
                addthread(clist, thread(pc->x));
```

```
                    break;
                case Split:
                    addthread(clist, thread(pc->x));
                    addthread(clist, thread(pc->y));
                    break;
                }
            }
            swap(clist, nlist);
            clear(nlist);
        }
    }
```

Suppose that there are *n* instructions in the regular expression program being run. Because the thread state is only the program counter, there are only *n* different possible threads that can appear on `clist` or `nlist`. If `addthread` does not add a thread to the list if an identical thread (with the same `pc`) is already on the list, then `ThreadLists` only need room for `n` possible threads, eliminating the possibility of overflow.

Having at most `n` threads on a list also bounds the amount of time spent processing each character. Assuming an efficient *O*(1) implementation of `addthread`, the worst case time for processing a single character is only *O*(*n*), so the time for the entire string is *O*(*n* *m*). This is far better than the essentially unbounded time requires by backtracking. (It also eliminates the infinite loops mentioned above.)

Strictly speaking, there's no reason why the backtracking VM implementation couldn't adopt the same trick, making sure not to queue a thread if an identical thread (with the same `pc` and `sp`) had already been queued. Doing so would require tracking *n* *m* possible threads: one for each possible `pc` and `sp` pair.

It is not uncommon to search for a 20-byte regular expression in a megabyte of text. In that case *n* is at most 40, but *n* *m* can be as high as 40 million. (And by today's standards, a megabyte of text is tiny!) The benefit of Thompson's approach is that, because the threads run in lock step, there are only *n* possible threads at a given point. The approach reducing the bookkeeping overhead dramatically by making it independent of the text length.

Tracking Submatches

Treating regular expressions as compiled to bytecodes makes it easy to add new features, like submatch tracking, by defining new bytecodes and implementing them.

To add submatch tracking, we'll add an array of saved string pointers to the thread state. The new bytecode instruction `save` *`i`* saves the current string pointer in the *i*th slot in the saved pointer array for the current thread. To compile `(`*e*`)`, which saves the boundaries of the match for *e*, we'll put `save` instructions around the code for *e*. For the *k*th set of parentheses (Perl's `$k`), we'll use slots 2*k* to hold the starting position and 2*k*+1 to hold the ending position.

For example, compare the compiled form of a+b+ and (a+)(b+):

```
          a+b+                (a+)(b+)

                          0   save 2
    0   char a            1   char a
    1   split 0, 2        2   split 1, 3
                          3   save 3
                          4   save 4
    2   char b            5   char b
    3   split 2, 4        6   split 5, 7
                          7   save 5
    4   match             8   match
```

If we wanted to find the boundaries of the entire match, we could wrap the generated bytecode in save 0 and save 1 instructions.

Implementing the save instruction in recursiveloop is straightforward (saved[pc->i]=sp) except that the assignment must be undone if the match goes on to fail. This insulates the successful thread from failed threads.

```
int
recursiveloop(Inst *pc, char *sp, char **saved)
{
    char *old;

    for(;;){
        switch(pc->opcode){
        case Char:
            if(*sp != pc->c)
                return 0;
            pc++;
            sp++;
            break;
        case Match:
            return 1;
        case Jmp:
            pc = pc->x;
            break;
        case Split:
            if(recursiveloop(pc->x, sp, saved))
                return 1;
            pc = pc->y;
            break;
        case Save:
            old = saved[pc->i];
            saved[pc->i] = sp;
            if(recursiveloop(pc+1, sp, saved))
                return 1;
            /* restore old if failed */
            saved[pc->i] = old;
            return 0;
```

```
                }
            }
        }
```

Notice that `case Save` has an unavoidable recursion, just like `case Split` does. The `Save` recursion is harder to compile out than the `Split` recursion; fitting `Save` into `backtrackingvm` requires more effort. The difference in effort is one reason implementors prefer recursion despite the possible stack overflow problems.

Pike's Implementation

In a "threaded" implementation like `thompsonvm` above, we simply add the saved pointer set to the thread state. Rob Pike first used this approach, in the text editor `sam`.

```
        struct Thread
        {
                Inst *pc;
                char *saved[20];  /* $0 through $9 */
        };
        Thread thread(Inst *pc, char **saved);
        int
        pikevm(Inst *prog, char *input, char **saved)
        {
            int len;
            ThreadList *clist, *nlist;
            Inst *pc;
            char *sp;
            Thread t;

            len = proglen(prog);  /* # of instructions */
            clist = threadlist(len);
            nlist = threadlist(len);

            addthread(clist, thread(prog, saved));
            for(sp=input; *sp; sp++){
                for(i=0; i>clist.n; i++){
                    t = clist.t[i];
                    switch(pc->opcode){
                    case Char:
                        if(*sp != pc->c)
                            break;
                        addthread(nlist, thread(t.pc+1, t.saved));
                        break;
                    case Match:
                        memmove(saved, t.saved, sizeof t.saved);
                        return 1;
                    case Jmp:
                        addthread(clist, thread(t.pc->x, t.saved));
                        break;
                    case Split:
                        addthread(clist, thread(t.pc->x, t.saved));
                        addthread(clist, thread(t.pc->y, t.saved));
                        break;
                    case Save:
                        t.saved[t->pc.i] = sp;
                        addthread(clist, thread(t.pc->x, t.saved));
                        break;
                    }
                }
                swap(clist, nlist);
                clear(nlist);
            }
        }
```

The `case Save` code is simpler in `pikevm` than in `recursiveloop`, because each thread has its own copy of `saved`: there is no need to restore the old value.

In Thompson's VM, `addthread` could limit the size of the thread lists to *n*, the length of the compiled program, by keeping only one thread with each possible PC. In Pike's VM, the thread state is larger—it includes the saved pointers too—but `addthread` can still keep just one thread with each possible PC. This is because the saved pointers do not influence future execution: they only record past execution. Two threads with the same PC will execute identically even if they have different saved pointers; thus only one thread per PC needs to be kept.

Ambiguous Submatching

Sometimes there is more than one way for a regular expression to match a string. As a simple example, consider searching for `<.*>` in `<html></html>`. Does the pattern match just `<html>` or the entire `<html></html>`? Equivalently, does the `.*` match just `html` or `html></html`? Perl, the de-facto standard for regular expression implementations, does the latter. In this sense, `*` is "greedy": it matches as much as it can while preserving the match.

Requiring `*` to be greedy essentially imposes a priority on each thread of execution. We can add such a priority to the VM specification by defining that the `split` instruction "prefers" successful execution using its first argument to its second argument.

With a prioritizing `split`, we can implement a greedy *e*`*` (and *e*`?` and *e*`+`) by making sure that the preferred choice is the one that matches more instances of *e*. Perl also introduced a "non-greedy" *e*`*?` (and *e*`??` and *e*`+?`) that matches as little as possible. It can be implemented by reversing the arguments to `split`, so that matching fewer instances is preferred.

The exact code sequences are:

	greedy (same as above)		non-greedy
e?	`split` **`L1, L2`**	*e*??	`split` **`L2, L1`**
`L1:`	*codes for e*	`L1:`	*codes for e*
`L2:`		`L2:`	

```
e*    L1: split L2, L3        e*?   L1: split L3, L2
      L2: codes for e               L2: codes for e
          jmp L1                        jmp L1
      L3:                           L3:
e+    L1: codes for e         e+?   L1: codes for e
          split L1, L3                  split L3, L1
      L3:                           L3:
```

The backtracking implementations given above already prioritize `split`'s choices in the way just defined, though verifying this fact takes a little effort. The `recursive` and `recursiveloop` implementations simply try `pc->x` before `pc->y`:

```
/* recursive */
case Split:
    if(recursive(pc->x, sp))
        return 1;
    return recursive(pc->y, sp);

/* recursiveloop */
case Split:
    if(recursiveloop(pc->x, sp))
        return 1;
    pc = pc->y;
    continue;
```

The `backtrackingvm` implementation creates a new thread for the lower-priority `pc->y` and continues executing at `pc->x`:

```
/* backtrackingvm */
case Split:
    if(nready >= MAXTHREAD){
        fprintf(stderr, "regexp overflow");
        return -1;
    }
    /* queue new thread */
    ready[nready++] = thread(pc->y, sp);
    pc = pc->x;  /* continue current thread */
    continue;
```

Because the threads are managed in a stack (last in, first out), the thread for `pc->y` will not execute until all threads created by `pc->x` have been tried and failed. Those threads all have higher priority than the `pc->y` thread.

The `pikevm` implementation given above does not completely respect thread priority, but it can be fixed. To make it respect priority, we can make `addthread` handle `Jmp`, `Split`, and `Save` instructions by calling itself recursively to add the targets of those instructions instead. (This makes `addthread` essentially identical to `addstate` in the first article.) This change ensures that `clist` and `nlist` are maintained in order of thread priority, from highest to lowest. The processing loop in `pikevm` thus tries threads in priority order, and the aggressive `addthread` makes sure that all threads generated from one priority level are added to `nlist` before considering threads from the next priority level.

The `pikevm` changes are motivated by the observation that recursion respects thread priority. The new code uses recursion while processing a single character, so that `nlist` will be generated in priority order, but it still advances threads in lock step to keep the good run-time behavior. Because `nlist` is generated in priority order, the "ignore a thread if the PC has been seen before" heuristic is safe: the thread seen earlier is higher priority and should be the one that gets saved.

There is one more change necessary in `pikevm`: if a match is found, threads that occur later in the `clist` (lower-priority ones) should be cut off, but higher-priority threads need to be given the chance to match possibly-longer sections of the string. The new main loop in `pikevm` looks like:

```
for(i=0; i<clist.n ;i++){
    pc = clist.t[i].pc;
    switch(pc->opcode){
    case Char:
        if(*sp != pc->c)
            break;
        addthread(nlist, thread(pc+1), sp+1);
        break;
    case Match:
        saved = t.saved;  // save end pointer
        matched = 1;
        clist.n = 0;
        break;
    }
}
```

The same changes can be made to `thompsonvm`, but since it does not record submatch locations, the only visible effect would be the VM's choice of end pointer when there is a match. The changes would make `thompsonvm` stop at the end of the same match that the backtracking implementation would have chosen. This will be useful in the next article.

An implementation can use other criteria to cull the thread set, by comparing the submatch sets directly. The Eighth Edition Unix library used leftmost longest as the criteria, to mimic DFA-based tools like awk and egrep.

Real world regular expressions

Regular expression usage in real programs is somewhat more complicated than what the regular expression implementations described above can handle. This section briefly describes how to accommodate a few common constructs.

Character classes. Character classes are typically implemented as a special VM instruction rather than expanding them into alternations. The sample code linked below implements a special case version of this, using an “any byte” instruction for the metacharacter `.` (dot).

Repetition. Repetitions often end at a character that cannot appear in the repetition. For example, in `/[0-9]+:[0-9]+/`, the first `[0-9]+` must end at `:` and the second must end at a non-digit. It is possible to introduce one-byte lookaheads to avoid the overhead of creating new threads during the repetition. This technique is most frequently seen in backtracking implementations, which would otherwise create one new thread per character. If threads are implemented recursively, then this optimization is necessary to avoid stack overflows for simple expressions.

Backtracking loops. At the beginning of the article, we noted that a naive backtracking algorithm could go into an infinite loop searching for `(a*)*`, because of the repeated empty match. A simple way to avoid these loops during backtracking is to introduce a progress instruction, which says that each time the machine executes that instruction it must have moved forward since the last time it was there.

Alternate program structures for backtracking. Another way to avoid loops is to change the instruction set to introduce instructions for the repetitions. The implementation of those instructions can call the subpieces as subroutines, which avoids the infinite loop problem and makes it more efficient to implement features like counted repetition and assertions. Even so, these constructs make the non-recursive implementation much harder (more state to keep around like the "old" pointer) and preclude the automata techniques behind Pike's VM implementation. It's also easy for the implementation to get out of control though: look at Perl or PCRE or really any other “full-featured” regexp implementation.

Backreferences. Backreferences are trivial in backtracking implementations. In Pike's VM, they can be accommodated, except that the argument about discarding threads with duplicate PCs no longer holds: two threads with the same PC might have different capture sets, and now the capture sets can influence future execution, so an implementation has to keep both threads, a potentially exponential blowup in state. GNU grep combines both approaches: it rewrites backreferences into an approximate regular expression that can be used with a DFA (for example, `(cat|dog)\1` becomes `(cat|dog)(cat|dog)`, which has a different, broader meaning than the original) and then the matches that the DFA turns up can be checked with the backtracking search.

Unanchored matches. To implement unanchored matches, many implementations try for a match at position 0, then try for a match at position 1, and so on. If the regexp engine scans to the end of the string before failing to find a match, wrapping it in this loop makes the unanchored search quadratic in the length of the text. In VM-based implementations, a more efficient way to do an unanchored search is to put the compiled code for `.*?` at the beginning of the program. This lets the VM itself implement the unanchored search in a single linear-time pass.

Character encodings. Since the Thompson and Pike VMs work a character at a time, never back up, and don't have any tables proportional to the size of the character set, it is easy to support alternate encodings and character sets, even UTF-8: just decode the character in the dispatch loop before running that step of the machine. These VMs have the property that they only decode each character once, which is nice when the decoding is complicated (UTF-8 decoding is not terribly expensive, but it's also not just a single pointer dereference).

Digression: POSIX Submatching

The POSIX committee decided that Perl's rules for resolving submatch ambiguity were too hard to explain, so they chose new rules that are easier to state but turn out to be harder to implement. (The rules also make it essentially impossible to define non-greedy operators and are incompatible with pre-existing regular expression implementations, so almost nobody uses them.) Needless to say, POSIX's rules haven't caught on.

POSIX defines that to resolve submatches, first chose the match that starts leftmost in the string. (This is traditional Perl behavior but here things diverge.) Among the submatches starting at the leftmost position in the string, choose the longest one overall. If there are still multiple choices, choose the ones that maximize the length of the leftmost element in the regular expression. Of the choices remaining, choose the ones that maximize the next element in the regular expression. And so on. In an NFA-based implementation, this requires putting extra parentheses around every concatenation expression of variable length. Then one can merge two threads by choosing the one that is "better" according to the POSIX rules. And then there are the rules for repetitions: POSIX defines that `x*` is like `(xx*)?`, where the first `x` is as long as possible, then the second one is, and so on. It is possible to invent contrived examples showing that an NFA must track every possible `x` in the repetition separately in order to merge threads correctly, so the amount of per-thread state required for a forward regular expression search that maintains states as described above is potentially unbounded.

For example, when when matching `(a|bcdef|g|ab|c|d|e|efg|fg)*` against `abcdefg`, there are three possible ways for the star operator to break up the string: `a bcdef g`, `ab c d efg`, and `ab c d e fg`. In Perl, the alternation prefers earlier alternatives, choosing the first version—using `a` in the first iteration trumps using `ab` because `a` is listed first—so the recorded submatch for those parentheses is `g`. In POSIX, the repetition prefers to match the largest possible substring in each step, which leads to the second version—using `ab` in the first repetition trumps `a`, and then using `efg` in the fourth iteration trumps `e`—so the recorded submatch for those parentheses is `efg`. Glenn Fowler has written a test suite for POSIX regular expression semantics, and Chris Kuklewicz has written a more complete test suite that finds bugs in most implementations.

There are two possible ways to avoid the seemingly unbounded tracking of space implied by POSIX submatching semantics. First, it turns out that matching the regular expression *backward* bounds the bookkeeping to being linear in the size of the regular expression. This program demonstrates the technique. Second, Chris Kuklewicz observes that it is possible to run the machine forward in bounded space if the machine regularly compares all threads, replacing the submatch data that would be used in the event of a collision with an assigned per-thread priority (from 1 to *n*). His regex-tdfa package for Haskell implements this technique.

Digression: A Forgotten Technique

The most interesting technique in this article is the one of storing submatch information in the regular expression thread state. The earliest instance I know of this technique in a regular expression engine is in Rob Pike's *sam* editor, written around 1985. (The modifications to store submatches were contributed by Bruce Janson a couple of years after the original implementation.) The technique makes a cameo in a textbook in 1974 but then seems to get lost until its reappearance in *sam*.

In Aho, Hopcroft, and Ullman's 1974 *The Design and Analysis of Computer Algorithms*, Chapter 9 is devoted to "Pattern Matching Algorithms." There are two interesting exercises in Chapter 9:

> 9.6 Let $x = a_1a_2...a_n$ be a given string and α a regular expression. Modify Algorithm 9.1 [the NFA simulation] to find the least k and, having found k, the (a) least j and (b) the greatest j such that $a_ja_{j+1}...a_k$ is in the set denoted by α. [Hint: Associate an integer j with each state in S_j.]
>
> *9.7 Let x and α be as in Exercise 9.6. Modify Algorithm 9.1 to find the least j, and having found j, the greatest k such that $a_ja_{j+1}...a_k$ is in the set denoted by α.

Exercise 9.6 is asking for the shortest match among those that end at the earliest possible string location. Exercise 9.7 is asking for the longest match among those that start at the earliest possible string location—the now standard "leftmost longest" match. In fact, text earlier in the chapter (in section 9.2) all but gave away the answer:

> Various pattern recognition algorithms may be constructed from Algorithm 9.1. For example, suppose we are given a regular expression α and a text string $x = a_1a_2...a_n$, and we wish to find the least k such that there exists a $j < k$ for which $a_ja_{j+1}...a_k$ is in the set denoted by α. Using Theorem 9.2 [the

> regular expression to NFA construction] we can construct from α an NFDFA M to accept the language $I^*\alpha$. To find the least k such that $a_1a_2...a_k$ is in $L(M)$, we can insert a test at the end of the block of lines 2-12 in Fig 9.11 [after each input character] to see whether S_i contains a state of F. We may, by Theorem 9.2, take F to be a singleton, so this test is not time-consuming; it is $O(m)$, where m is the number of states in M. If S_i contains a state of F, then we break out of the main loop, having found $a_1a_2...a_i$ to be the shortest prefix of x in $L(M)$.
>
> Algorithm 9.1 can be further modified to produce for each such k the greatest $j < k$ (or the least j) such that $a_ja_{j+1}...a_k$ is in the set denoted by α. This is done by associating an integer with each state in the sets S_i. The integer associated with state s in S_k indicates the greatest j (or least j) such that $(s_0, a_ja_{j+1}...a_k) \vdash^* (s, \varepsilon)$. The details of updating these integers in Algorithm 9.1 are left as an exercise.

As far as I can tell, the technique hinted at in the chapter and in the exercise was forgotten until its use in *sam*, and even then went unnoticed for over a decade. In particular, neither the exercise nor the technique make any appearance in Aho, Hopcroft, or Ullman's later textbooks, and the regular expression matcher in *awk* (Aho is the *a* in *awk*) uses the naive quadratic “loop around a DFA” to implement leftmost-longest matching.)

Digression: “Thompson's Algorithm”

The first article in this series explains:

> R. McNaughton and H. Yamada and Ken Thompson are commonly credited with giving the first constructions to convert regular expressions into NFAs, even though neither paper mentions the then-nascent concept of an NFA. McNaughton and Yamada's construction creates a DFA, and Thompson's construction creates IBM 7094 machine code, but reading between the lines one can see latent NFA constructions underlying both.

It's interesting to trace the history of how an NFA construction has come to be credited to two papers that didn't mention NFAs. Thompson's paper's only reference to theory is the sentence “In the terms of Brzozowski [1], this algorithm continually takes the left derivative of the given regular expression with respect to the text to be searched.”

It was Aho, Hopcroft, and Ullman's 1974 textbook *The Design and Analysis of Computer Algorithms* that first presented the algorithm in terms of automata, converting the regular expression to an NFA and then giving an algorithm to execute the NFA. The Bibliographic Notes at the end of the chapter explain that “The regular expression pattern-matching algorithm (Algorithm 9.1) is an abstraction of an algorithm by Thompson [1968].”

Aho and Ullman's 1977 *Principles of Compiler Design* presents the algorithm to convert regular expressions to NFAs without an attribution. A later discussion about executing regular expressions reads:

> The time we spend processing the regular-expression pattern to convert it to a form, such as a DFA, suitable for scanning the input line could far exceed the actual time spent scanning the line.
>
> To make the whole process efficient we must balance the time spent processing the pattern and doing the scanning. A reasonable compromise, proposed by Thompson [1968], is to convert the regular expression to an NFA. The process of scanning the input is then one of directly simulating the NFA. As we scan the line, a list of “current” states is kept, beginning with the ε-CLOSURE of the start state. If a is the next input character, we create a new list of all states with a transition on a from a state of the old list. The old list is then discarded and we compute the ε-CLOSURE of the new list. If the final state is not on the new list, we repeat the process with the next character.

The Bibliographic Notes say “Thompson [1968] describes a regular-expression recognition algorithm used in the QED text editor.”

Aho, Sethi, and Ullman's 1986 *Compilers: Principles, Techniques, and Tools* (the Dragon Book) gives the algorithm for converting an NFA to a regular expression as “Algorithm 3.3. (Thompson's construction).” Algorithm 3.4, “Simulating an NFA,” is uncredited. The Bibliographic Notes read “Many text editors use regular expressions for context searches. Thompson [1968], for example, describes the construction of an NFA from a regular expression (Algorithm 3.3) in the context of the QED text editor.” Where the 1974 textbook referred to the construction as “an abstraction of an algorithm by Thompson,” the Dragon book simply calls it “Thompson's construction.” The two are easily distinguished. If you read past the machine code to look at the NFA underneath, Thompson's 1968 paper uses one NFA state per literal character, union, or star. In contrast, the Dragon book presentation uses two states per character, union, or star and then drops one state per concatenation (the 1974 presentation was similar but without the concatenation optimization). The introduction of all the extra states makes the proofs easier but can double the number of states to be managed in a practical implementation. I have found many papers that use the Dragon book construction but cite Thompson's paper instead. I've even seen papers that “optimize” Thompson's construction, presenting algorithms that post-process the Dragon book algorithm to remove the unnecessary states.

Aho and Ullman's 1992 *Foundations of Computer Science* devotes Chapter 10 to "Patterns, Automata, and Regular Expressions," giving the now familiar constructions without attribution in the main text. The Bibliographic Notes read "The construction of nondeterministic automata from regular expressions that we used in Section 10.8 is from McNaughton and Yamada [1960].... The use of regular expressions as a way to describe patterns in strings first appeared in Ken Thompson's QED system (Thompson [1968]), and the same ideas later influenced many commands in his UNIX system." The reference to McNaughton and Yamada's paper is interesting because, like Thompson's, their paper does not mention NFAs either. It does give an algorithm for constructing a DFA from a regular expression, and that DFA is quite clearly (in hindsight) the result of applying the subset construction to the usual NFA, but the NFA itself is never mentioned.

Hopcroft, Motwani, and Ullman's 2001 *Introduction to Automata Theory, Languages, and Computation* also gives the familiar constructions. The References for Chapter 3 reads: "[The] construction of an ε-NFA from a regular expression, as presented here, is the 'McNaughton-Yamada construction,' from [McNaughton and Yamada, 1960].... Even before developing UNIX, K. Thompson was investigating the use of regular expressions in commands such as `grep`, and his algorithm for processing such commands appears in [Thompson, 1968]." The 2007 revision of the textbook left this section unchanged.

Aho, Lam, Sethi, and Ullman's 2007 *Compilers: Principles, Techniques, and Tools (2nd Edition)* gives by far the most accurate account of the history. Where the 1986 book described Algorithm 3.3 as simply "Thompson's construction," the 2007 edition describes the same algorithm (now Algorithm 3.23) as "The McNaughton-Yamada-Thompson algorithm to convert a regular expression to an NFA." Then the References for Chapter 3 elaborate:

> McNaughton and Yamada [1960] first gave an algorithm to convert regular expressions directly to finite automata. Algorithm 3.36 described in Section 3.9 was first used by Aho in creating the Unix regular-expression matching tool `egrep`. This algorithm was also used in the regular-expression pattern matching routines in `awk` [Aho, Kernighan, and Weinberger, 1988]. The approach of using nondeterministic automata as intermediary is due Thompson [1968]. The latter paper also contains the algorithm for direct simulation of nondeterministic finite automata (Algorithm 3.22), which was used by Thompson in the text editor `QED`.

By the way, Brzozowski's approach, cited by Thompson, fell out of favor and was largely ignored for many years, but Owens, Reppy, and Turon's excellent paper "Regular-expression derivatives reexamined" (JFP, 19(2), March 2009) points out that the approach works even better than automata in languages with good support for symbolic manipulation.

Implementations

See the previous article for detailed history. The source code for this article is available at http://code.google.com/p/re1/.

For a tour through the guts of a modern backtracking engine, see Henry Spencer's chapter "A Regular-Expression Matcher" in the book *Software Solutions In C*, Dale Schumacher, ed., Academic Press, 1994.

Summary

Thinking about regular expressions as programs for a virtual machine is a useful abstraction: a single regular expression parser can compile the regular expression into byte codes, and then different implementations can be used to execute the byte codes, depending on the context. Automata-based implementations can track submatch boundaries and get the same answers as a traditional backtracking implementation, with a guaranteed linear run time.

Thanks to Alex Healy and Chris Kuklewicz for helpful discussions about Perl and POSIX submatch semantics.

P.S. If you liked reading this, you might also be interested to read Roberto Ierusalimschy's paper "A Text Pattern-Matching Tool based on Parsing Expression Grammars," which uses a similar approach to match PEGs.

The next article in this series is "Regular Expression Matching in the Wild," a tour of a production implementation.

References

[1] Rob Pike, "The text editor sam," Software—Practice & Experience 17(11) (November 1987), pp. 813–845. *http://plan9.bell-labs.com/sys/doc/sam/sam.html*

[2] Ken Thompson, "Regular expression search algorithm," Communications of the ACM 11(6) (June 1968), pp. 419–422. *http://doi.acm.org/10.1145/363347.363387* (PDF)