Implementing Regular Expressions

CS 121 Extra Lecture
November 17, 2000

Note: These slides were used for a supplemental lecture in Harvard's introductory theory of computation course. The topic of the lecture was implementing regular expressions a la Ken Thompson. I quite enjoyed preparing and giving the talk, and I believe the other teaching fellows who attended enjoyed listening to it. I'm not sure that many of the students made it through to the end, though. Without the accompanying narrative, you'll definitely want a copy of the original paper at hand.

Perhaps the most useful part of these slides is the explanation of what the instructions actually mean, as IBM 7094 information is ever harder to find!

A newer version of this talk, in written form, is at `http://swtch.com/~rsc/regexp/`. It builds the machines using C data structures instead of IBM 7094 machine code. Perhaps some day I will build one using a more recent machine code. Also at that URL is a standalone version of the IBM 7094 cheat sheet.

The original paper twice contains the instruction `SCA NNODE, 0`, which must behave as though `XR[0]` is an alias for the constant zero. After consulting with Ken Thompson, I originally believed it was a typo. However, looking at an actual copy of Thompson's QED in the CTSS sources suggests that it is not a typo but actually an undocumented use of the `SCA` instruction.

Russ Cox
rsc@swtch.com
November 2005
January 2007 (updated text above)

Outline

History

Uses of regular expressions

Thompson’s algorithm in detail

Other algorithms in brief

History

Finite automata first introduced by McCullock and Pitts (1943) to model neurons.

> ‘‘A logical calculus of ideas immanent in nervous activity,’’ *Bull. Math. Biophys.* 5, pp. 115-133.

Formalized and cleaned up by Kleene.

> ‘‘Representation of events in nerve nets and finite automata’’, *Rand Research Memorandum*, RM-704, 1951. Also in *Automata Studies* (Princeton UP, 1956) pp. 129-156.

Kleene proved that regular expressions and finite automata describe the same languages.

Uses of Regular Expressions

Lots of text processing.

Most editors, scripting languages.

Mail routing.

Spam filtering.

When you’re processing a *lot* of text, you have to do it quickly.

Regular Expression Search Algorithm

Ken Thompson, ‘‘Regular Expression Search Algorithm’’, *CACM* 11(6), June 1968, pp. 419-422.

The algorithm:

First, check regular expression for syntax, inserting `.` for implicit concatenations.

Second, transform the regular expression into postfix notation, using `.` for concatenation.

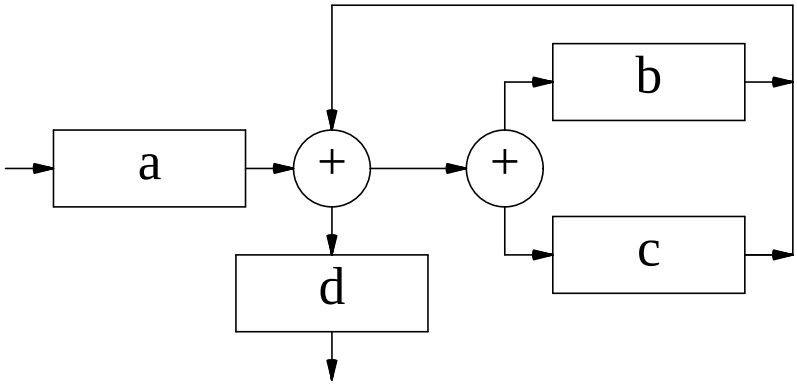
`a(b|c)*d` becomes `abc|*.d.`

Third, using a pushdown stack, put together an NDFA for the regular expression.

The nodes in the NDFA are little pieces of machine code!

Creating the NDFA

Follow the construction given in L&P, really.



Unlabeled nodes are *e*-transitions.

A digression: the IBM 7094

Popular high-end computer in the early 1960s.

Transistorized version of IBM 709.

Could add floating point numbers at 0.35 MIPS.

Cost approximately $3.5 million.

Base machine for the early time-shared operating system CTSS at MIT.

In some sense, CTSS begat Multics, which begat Unix.

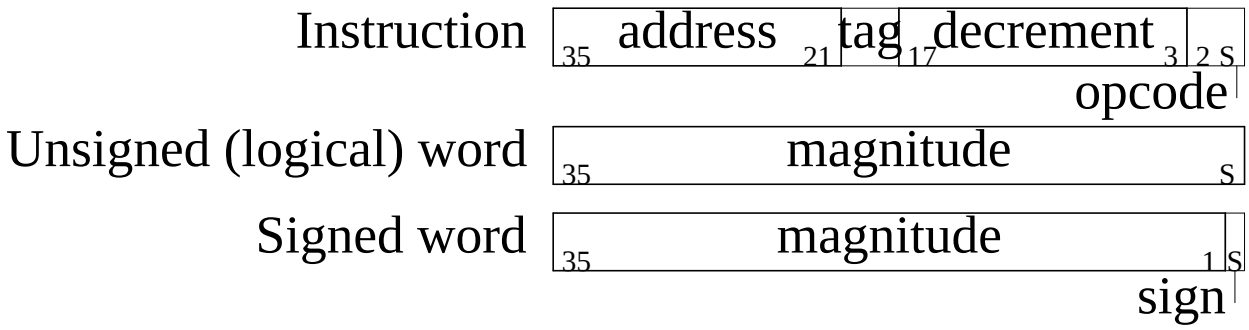
IBM 7094: Machine Layout

38-bit Accumulator (AC): 35 bit magnitude,
sign bit, extra bit for unsigned words, carry bit.
denote logical word by `AC1`, signed word by `ACs`

Memory (`M[addr]`): 32,768 36-bit words in main core.

Seven 15-bit index registers (`XR[i]`): usually held pointers.

Word layouts:



IBM 7094: Instruction Set

ACL *addr* ‘‘add and carry logical word’’
AC ← AC + M[*addr*]

AXC *addr*, *index* ‘‘address to index, complement’’
XR[*index*] ← *addr*

CAL *addr*, *index* ‘‘clear and add, logical’’
AC ← 0; ACl ← ACl+M[*addr*+XR[*index*]]

CLA *addr* ‘‘clear and add’’
AC ← 0; ACs ← ACs+M[*addr*]

LAC *addr*, *index* ‘‘load complement of address in index’’
XR[*index*] ← 2^{15} − M[*addr*]

IBM 7094: Instruction Set, II

PAC , *index* ‘‘place complement of address in index’’
XR[*index*] ← 2^{15} – AC<35:21>

PCA , *index* ‘‘place complement of index in address’’
AC ← 0; AC<35:21> ← 2^{15} – XR[*index*]

SCA *addr*, *index* ‘‘store complement of index in address’’
M[*addr*] ← 2^{15} – XR[*index*]

SLW *addr*, *index* ‘‘store logical word’’ (see CAL)
M[*addr*+XR[*index*]] ← AC⌉

IBM 7094: Instruction Set, III: Transfers

`TRA` *label*`,` *index* ‘‘transfer’’ (branch, jump)
`IC` ← *label*+`XR[`*index*`]`

`TSX` *label*`,` *index* ‘‘transfer and set index’’ (call)
`XR[`*index*`]` ← 2^{15} `- IC; IC` ← *label*

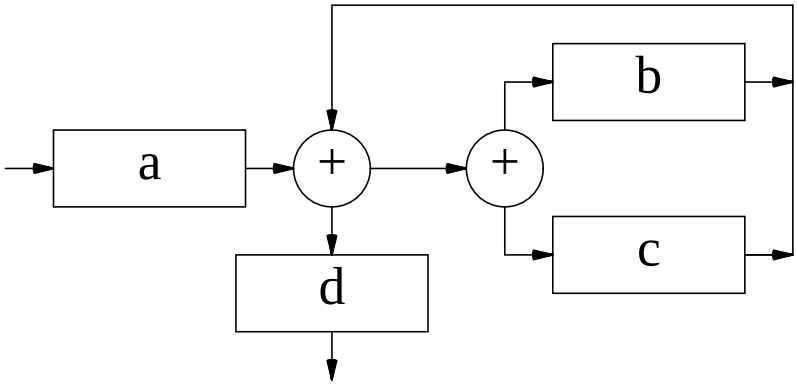
`TXI` *label*, *index*, *decr* ‘‘transfer with index incremented’’
`XR[`*index*`]` ← `XR[`*index]* + *decr; IC* ← *label*

`TXH` *label*`,` *index*`,` *decr* ‘‘transfer on index high’’
`if (decr < XR[`*index*`]) IC` ← *label*

`TXL` *label*`,` *index*`,` *decr* ‘‘transfer on index low’’
`if (decr >= XR[`*index*`]) IC` ← *label*

Creating the NDFA

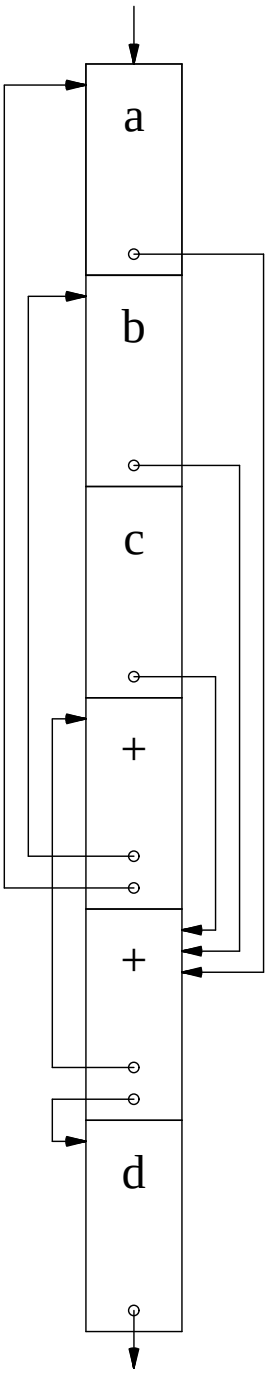
Follow the construction given in L&P, really.



Unlabeled nodes are *e*-transitions.

[This slide is identical to one before, just for your memory.]

The NDFA as machine code




```
 0 TRA CODE+1

 1 TXL FAIL,1–'a'–1
   TXH FAIL,1–'a'
   TSX NNODE,4
   TRA CODE+16

 5 TXL FAIL,1–'b'–1
   TXH FAIL,1–'b'
   TSX NNODE,4
   TRA CODE+16

 9 TXL FAIL,1–'c'–1
   TXH FAIL,1–'c'
   TSX NNODE,4
   TRA CODE+16

13 TSX CNODE,4
   TRA CODE+9
   TRA CODE+5

16 TSX CNODE,4
   TRA CODE+13
   TRA CODE+19

19 TXL FAIL,1–'d'–1
   TXH FAIL,1–'d'
   TSX NNODE,4
   TRA FOUND
```

Reading the Machine Code NDFA

Read the `TSX NNODE,4` instructions as ‘‘get a new letter’’.

Read the `TSX CNODE,4` instructions as ‘‘non-deterministically choose one of the next two instructions’’.

For each execution path from start to the end, you can read the letters off the path to form a string generated by the regular expression.

That’s basically what happens to do the match, except we run the machine one step at a time, and keep track of all possible states.

Running the Machine Code NDFA

We keep a list `CLIST` representing the current state set.

For each letter read, we process `CLIST` to produce `NLIST`, the next state set. Then we copy `CLIST=NLIST` and empty `NLIST`.

Each state on the list is represented by a `TSX CODE+`*n* instruction, a ‘‘function call’’ into the machine code NDFA.

`TSX NNODE,4` really means ‘‘append a jump to the next instruction onto `NLIST` and then return’’.

`TSX CNODE,4` really means ‘‘append a jump to the next instruction onto `CLIST` and then continue with the instruction after that’’.

`TRA XCHG` instruction terminates `CLIST`.

`XCHG` copies `NLIST` to `CLIST`, grabs the next letter, starts the next step.

Running the NDFA on abdx: a, b part i

Now CLIST=<TSX CODE+0,2; TRA XCHG>. Load a into XR[1].

Running CODE+0 returns when we call NNODE to add TSX CODE+4,2 to NLIST.

Now CLIST=<TSX CODE+4,2; TRA XCHG>. Load b into XR[1].

Run TSX CODE+4,2, jump to 16: we call CNODE.

CNODE adds TSX CODE+17,2 to CLIST, returns to 18.

At 18*ff*, we don't have a d, so we jump to FAIL, return to CLIST.

Now CLIST=<TSX CODE+0,2; TSX CODE+17,2; TRA XCHG> and we've just returned from the first call.

Running the NDFA on abdx: b part ii

Run TSX CODE+17,2, jump to 13: we call CNODE.

CNODE adds TSX CODE+14,2 to CLIST, returns to 15.

15 jumps to 5, we don't have a b, jump to FAIL, return to CLIST.

Now CLIST=<TSX CODE+0,2; TSX CODE+17,2;
TSX CODE+14,2; TRA XCHG>
and we've just returned from the second call.

Run TSX CODE+14,2, jump to 9, we don't have a c, jump to FAIL, return to CLIST.

Now CLIST=<TSX CODE+0,2; TSX CODE+17,2;
TSX CODE+14,2; TRA XCHG>
and we've just returned from the third call.

NLIST=<TSX CODE+12,2>, copy it to CLIST, on to d.

Running the NDFA on abdx: d, x

Now CLIST=<TSX CODE+12,2; TRA XCHG>. Load d into XR[1].

Run TSX CODE+12,2, jump to 16, do what we did for b.

We have a d so call NNODE from 21 and fail elsewhere.

Now CLIST=<TSX CODE+22,2; TRA XCHG>. Load x into XR[1].

Run TSX CODE+22,2, jump to FOUND. Match!

Note that match signal was delayed one letter.

NNODE

```
TSXCMD TSX 1,2          return to one instruction past caller
```

To be used later.

```
NNODE  AXC **,7         XR[7] ← n
```

Note that *n* is a ‘constant’ in the first instruction.

```
       PCA ,4           AC<21:35> ← 2^15−XR[4]
       ACL TSXCMD       AC⁀ ← AC⁀+M[TSXCMD]
       SLW NLIST,7      M[NLIST+XR[7]] ← AC⁀
```

Store `TSX` *ra+1*`,2` into `NLIST[`*n*`]`.

```
       TXI *+1,7,-1     XR[7] −= −1; ‘‘jump’’ to next line
       SCA NNODE,7      M[NNODE]<addr bits> ← XR[7]
```

Increment *n* and store it *back into the instruction stream*.

```
       TRA 1,2
```

Return to `CODE`*’s caller*.

Note ‘‘calling conventions’’: calls from `CODE` put return address in `XR[4]`, calls from `XCHG` put return address in `XR[2]`. *No stack.*

FAIL

```
FAIL    TRA 1,2
```

Return to CODE*’s caller*.

CNODE

```
CNODE  AXC **,7          XR[7] ← **
       CAL CLIST,7       AC⌋ ← M[CLIST+XR[7]]
       SLW CLIST+1,7     M[CLIST+1+XR[7]] ← AC⌋
```

Effectively CLIST[*n*+1] ← CLIST[*n*].

```
       PCA ,4            AC<35:21> ← 2^15–XR[4]
       ACL TSXCMD        AC⌋ ← AC⌋+M[TSXCMD]
       SLW CLIST,7       M[CLIST+XR[7]] ← AC⌋
```

Store TSX *ra*,2 into CLIST[*n*].

```
       TXI *+1,7,-1      XR[7] -= -1; ‘‘jump’’ to next line
       SCA CNODE,7       M[CNODE]<addr bits> ← XR[7]
```

Increment *n* in the instruction stream.

```
       TRA 2,4
```

Return to two instructions past our caller.

Remember calling conventions: calls from CODE put return address in XR[4], calls from XCHG put return address in XR[2].

XCHG, I

```
TRACMD TRA XCHG
```

For later.

```
XCHG   LAC NNODE,7      XR[7] ← n (from NNODE)
       AXC 0,6          XR[6] ← 0
  X1   TXL X2,7,0       if (X[7] <= 0) goto X2
       TXI *+1,7,1      XR[7] -= 1
       CAL NLIST,7      AC⌋ ← M[NLIST+XR[7]]
       SLW CLIST,6      M[CLIST+XR[6]] ← AC⌋
       TXI X1,6,-1      XR[6] += 1; goto X1
```

Copy NLIST onto CLIST, note that it reverses.

```
  X2   CAL TRACMD       AC⌋ ← M[TRACMD]
       SLW CLIST,6      CLIST[XR[6]] ← AC⌋
       SCA CNODE,6      n from CNODE ← XR[6]
       SCA NNODE,0      n from NNODE ← 0 (how?)
```

Terminate CLIST with TRA XCHG, store list counts into instruction stream.

XCHG, II

```
TSX GETCHA,4
PAC ,1              XR[1] ← 2^15−AC<35:21>
```

Fetch next character.

```
TSX CODE,2
TRA CLIST
```

Start new search here, continue old search.

INIT

```
INIT   SCA NNODE,0
       TRA XCHG
```

Set up empty NLIST, jump into the exchanger.

Points of note

No stack. (Explicit return address registers.)

No loop. (Explicit `TRA XCHG` added to `CLIST`.)

Hardly ‘‘reentrant’’. (Modification of instruction stream.)

Backreferences

With backreferences, `(.*)\1` matches `catcat` or `dogdog` but not `dogcat`.

No longer describing a regular language, as you showed earlier in the course.

Matching with backreferences takes exponential time in all current implementations.

Matching with backreferences is NP-complete, a fact you will probably prove!

SGI's `sed` sucks even without backreferences:

```
abcdefghi abcdefgh abcdefg abcdef abcde abcd abc ab a
abcdefghi abcdefgh abcdefg abcdef abcde abcd abc ab a
```

Match 1000 lines of this against `.*a*.*`, `.*a*.*b*.*`, etc.

Takes 0.5 seconds for the first, 41 seconds for `.*a.*b.*c.*d.*e.*f.*g.*h.*i.*`.

Seems to be about 1.7^n seconds where *n* is the number of letters.

Current Implementations

When performance really matters, compile to a DFA: `awk` (855 lines of C), `lex`.

Some editors still use Thompson's NDFA simulation (`sam`, `acme`).

If you want to know which pieces of the matched text were matched by various parts of the regexp, it's not hard to augment the lists with ‘‘parentheses’’ information.

Many scripting languages (`Perl`, `sed`, etc.) and editors (`vi`, `emacs`) run an exhaustive search on the paths through the NDFAs. This allows backreferences and other irregularities.

Tcl seems to combine all of these and more, picking a good one for the task at hand. (8677 lines of C).