NAME

perlreguts - Description of the Perl regular expression engine.

DESCRIPTION

This document is an attempt to shine some light on the guts of the regex engine and how it works. The regex engine represents a significant chunk of the perl codebase, but is relatively poorly understood. This document is a meagre attempt at addressing this situation. It is derived from the author's experience, comments in the source code, other papers on the regex engine, feedback on the perl5-porters mail list, and no doubt other places as well.

WARNING! It should be clearly understood that this document represents the state of the regex engine as the author understands it at the time of writing. It is **NOT** an API definition; it is purely an internals guide for those who want to hack the regex engine, or understand how the regex engine works. Readers of this document are expected to understand perl's regex syntax and its usage in detail. If you want to learn about the basics of Perl's regular expressions, see *perlre*.

OVERVIEW

A quick note on terms

There is some debate as to whether to say "regexp" or "regex". In this document we will use the term "regex" unless there is a special reason not to, in which case we will explain why.

When speaking about regexes we need to distinguish between their source code form and their internal form. In this document we will use the term "pattern" when we speak of their textual, source code form, the term "program" when we speak of their internal representation. These correspond to the terms *S-regex* and *B-regex* that Mark Jason Dominus employs in his paper on "Rx" ([1] in *REFERENCES*).

What is a regular expression engine?

A regular expression engine is a program that takes a set of constraints specified in a mini-language, and then applies those constraints to a target string, and determines whether or not the string satisfies the constraints. See *perlre* for a full definition of the language.

So in less grandiose terms the first part of the job is to turn a pattern into something the computer can efficiently use to find the matching point in the string, and the second part is performing the search itself.

To do this we need to produce a program by parsing the text. We then need to execute the program to find the point in the string that matches. And we need to do the whole thing efficiently.

Structure of a Regexp Program

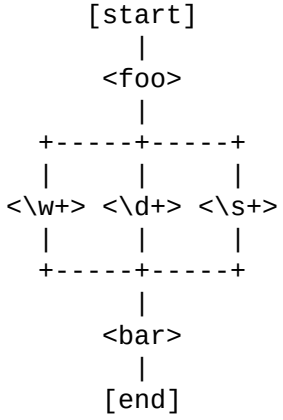
High Level

Although it is a bit confusing and some people object to the terminology, it is worth taking a look at a comment that has been in *regexp.h* for years:

This is essentially a linear encoding of a nondeterministic finite-state machine (aka syntax charts or "railroad normal form" in parsing technology).

The term "railroad normal form" is a bit esoteric, with "syntax diagram/charts", or "railroad diagram/charts" being more common terms. Nevertheless it provides a useful mental image of a regex program: each node can be thought of as a unit of track, with a single entry and in most cases a single exit point (there are pieces of track that fork, but statistically not many), and the whole forms a layout with a single entry and single exit point. The matching process can be thought of as a car that moves along the track, with the particular route through the system being determined by the character read at each possible connector point. A car can fall off the track at any point but it may only proceed as long as it matches the track.

Thus the pattern `/foo(?:\w+|\d+|\s+)bar/` can be thought of as the following chart:



The truth of the matter is that perl's regular expressions these days are much more complex than this kind of structure, but visualising it this way can help when trying to get your bearings, and it matches the current implementation pretty closely.

To be more precise, we will say that a regex program is an encoding of a graph. Each node in the graph corresponds to part of the original regex pattern, such as a literal string or a branch, and has a pointer to the nodes representing the next component to be matched. Since "node" and "opcode" already have other meanings in the perl source, we will call the nodes in a regex program "regops".

The program is represented by an array of `regnode` structures, one or more of which represent a single regop of the program. Struct `regnode` is the smallest struct needed, and has a field structure which is shared with all the other larger structures.

The "next" pointers of all regops except `BRANCH` implement concatenation; a "next" pointer with a `BRANCH` on both ends of it is connecting two alternatives. [Here we have one of the subtle syntax dependencies: an individual `BRANCH` (as opposed to a collection of them) is never concatenated with anything because of operator precedence.]

The operand of some types of regop is a literal string; for others, it is a regop leading into a sub-program. In particular, the operand of a `BRANCH` node is the first regop of the branch.

NOTE: As the railroad metaphor suggests, this is **not** a tree structure: the tail of the branch connects to the thing following the set of `BRANCH`es. It is a like a single line of railway track that splits as it goes into a station or railway yard and rejoins as it comes out the other side.

Regops

The base structure of a regop is defined in *regexp.h* as follows:

```
struct regnode {
    U8  flags;    /* Various purposes, sometimes overridden */
    U8  type;     /* Opcode value as specified by regnodes.h */
    U16 next_off; /* Offset in size regnode */
};
```

Other larger `regnode`-like structures are defined in *regcomp.h*. They are almost like subclasses in that they have the same fields as `regnode`, with possibly additional fields following in the structure, and in some cases the specific meaning (and name) of some of base fields are overridden. The following is a more complete description.

`regnode_1`

`regnode_2`

`regnode_1` structures have the same header, followed by a single four-byte argument; `regnode_2` structures contain two two-byte arguments instead:

```
    regnode_1               U32 arg1;
    regnode_2               U16 arg1;  U16 arg2;
```

`regnode_string`

`regnode_string` structures, used for literal strings, follow the header with a one-byte length and then the string data. Strings are padded on the end with zero bytes so that the total length of the node is a multiple of four bytes:

```
    regnode_string          char string[1];
                            U8 str_len; /* overrides flags */
```

`regnode_charclass`

Character classes are represented by `regnode_charclass` structures, which have a four-byte argument and then a 32-byte (256-bit) bitmap indicating which characters are included in the class.

```
    regnode_charclass       U32 arg1;
                            char bitmap[ANYOF_BITMAP_SIZE];
```

`regnode_charclass_class`

There is also a larger form of a char class structure used to represent POSIX char classes called `regnode_charclass_class` which has an additional 4-byte (32-bit) bitmap indicating which POSIX char class have been included.

```
    regnode_charclass_class U32 arg1;
                            char bitmap[ANYOF_BITMAP_SIZE];
                            char classflags[ANYOF_CLASSBITMAP_SIZE];
```

regnodes.h defines an array called `regarglen[]` which gives the size of each opcode in units of `size regnode` (4-byte). A macro is used to calculate the size of an `EXACT` node based on its `str_len` field.

The regops are defined in *regnodes.h* which is generated from *regcomp.sym* by *regcomp.pl*. Currently the maximum possible number of distinct regops is restricted to 256, with about a quarter already used.

A set of macros makes accessing the fields easier and more consistent. These include `OP()`, which is used to determine the type of a `regnode`-like structure; `NEXT_OFF()`, which is the offset to the next node (more on this later); `ARG()`, `ARG1()`, `ARG2()`, `ARG_SET()`, and equivalents for reading and setting the arguments; and `STR_LEN()`, `STRING()` and `OPERAND()` for manipulating strings and regop bearing types.

What regop is next?

There are three distinct concepts of "next" in the regex engine, and it is important to keep them clear.

- There is the "next regnode" from a given regnode, a value which is rarely useful except that sometimes it matches up in terms of value with one of the others, and that sometimes the code assumes this to always be so.
- There is the "next regop" from a given regop/regnode. This is the regop physically located after the the current one, as determined by the size of the current regop. This is often useful, such as when dumping the structure we use this order to traverse. Sometimes the code assumes that the "next regnode" is the same as the "next regop", or in other words assumes that the sizeof a given regop type is always going to be one regnode large.
- There is the "regnext" from a given regop. This is the regop which is reached by jumping forward by the value of `NEXT_OFF()`, or in a few cases for longer jumps by the `arg1` field of

the regnode_1 structure. The subroutine regnext() handles this transparently. This is the logical successor of the node, which in some cases, like that of the BRANCH regop, has special meaning.

Process Overview

Broadly speaking, performing a match of a string against a pattern involves the following steps:

A. Compilation

1. Parsing for size
2. Parsing for construction
3. Peep-hole optimisation and analysis

B. Execution

4. Start position and no-match optimisations
5. Program execution

Where these steps occur in the actual execution of a perl program is determined by whether the pattern involves interpolating any string variables. If interpolation occurs, then compilation happens at run time. If it does not, then compilation is performed at compile time. (The /o modifier changes this, as does qr// to a certain extent.) The engine doesn't really care that much.

Compilation

This code resides primarily in *regcomp.c*, along with the header files *regcomp.h*, *regexp.h* and *regnodes.h*.

Compilation starts with pregcomp(), which is mostly an initialisation wrapper which farms work out to two other routines for the heavy lifting: the first is reg(), which is the start point for parsing; the second, study_chunk(), is responsible for optimisation.

Initialisation in pregcomp() mostly involves the creation and data-filling of a special structure, RExC_state_t (defined in *regcomp.c*). Almost all internally-used routines in *regcomp.h* take a pointer to one of these structures as their first argument, with the name pRExC_state. This structure is used to store the compilation state and contains many fields. Likewise there are many macros which operate on this variable: anything that looks like RExC_xxxx is a macro that operates on this pointer/structure.

Parsing for size

In this pass the input pattern is parsed in order to calculate how much space is needed for each regop we would need to emit. The size is also used to determine whether long jumps will be required in the program.

This stage is controlled by the macro SIZE_ONLY being set.

The parse proceeds pretty much exactly as it does during the construction phase, except that most routines are short-circuited to change the size field RExC_size and not do anything else.

Parsing for construction

Once the size of the program has been determined, the pattern is parsed again, but this time for real. Now SIZE_ONLY will be false, and the actual construction can occur.

reg() is the start of the parse process. It is responsible for parsing an arbitrary chunk of pattern up to either the end of the string, or the first closing parenthesis it encounters in the pattern. This means it can be used to parse the top-level regex, or any section inside of a grouping parenthesis. It also handles the "special parens" that perl's regexes have. For instance when parsing /x(?:foo)y/ reg() will at one point be called to parse from the "?" symbol up to and including the ")".

Additionally, reg() is responsible for parsing the one or more branches from the pattern, and for

"finishing them off" by correctly setting their next pointers. In order to do the parsing, it repeatedly calls out to `regbranch()`, which is responsible for handling up to the first `|` symbol it sees.

`regbranch()` in turn calls `regpiece()` which handles "things" followed by a quantifier. In order to parse the "things", `regatom()` is called. This is the lowest level routine which parses out constant strings, character classes, and the various special symbols like `$`. If `regatom()` encounters a "(" character it in turn calls `reg()`.

The routine `regtail()` is called by both `reg()`, `regbranch()` in order to "set the tail pointer" correctly. When executing and we get to the end of a branch, we need to go to the node following the grouping parens. When parsing, however, we don't know where the end will be until we get there, so when we do we must go back and update the offsets as appropriate. `regtail` is used to make this easier.

A subtlety of the parsing process means that a regex like `/foo/` is originally parsed into an alternation with a single branch. It is only afterwards that the optimiser converts single branch alternations into the simpler form.

Parse Call Graph and a Grammar

The call graph looks like this:

```
    reg()                       # parse a top level regex, or inside of
parens
        regbranch()             # parse a single branch of an alternation
            regpiece()          # parse a pattern followed by a quantifier
                regatom()       # parse a simple pattern
                    regclass()  #   used to handle a class
                    reg()       #   used to handle a parenthesised
subpattern
                    ....
            ...
            regtail()           # finish off the branch
        ...
        regtail()               # finish off the branch sequence. Tie each
                                # branch's tail to the tail of the
sequence
                                # (NEW) In Debug mode this is
                                # regtail_study().
```

A grammar form might be something like this:

```
    atom  : constant | class
    quant : '*' | '+' | '?' | '{min,max}'
    _branch: piece
           | piece _branch
           | nothing
    branch: _branch
          | _branch '|' branch
    group : '(' branch ')'
    _piece: atom | group
    piece : _piece
          | _piece quant
```

Debug Output

In the 5.9.x development version of perl you can `use re Debug => 'PARSE'` to see some trace information about the parse process. We will start with some simple patterns and build up to more complex patterns.

So when we parse `/foo/` we see something like the following table. The left shows what is being parsed, and the number indicates where the next regop would go. The stuff on the right is the trace output of the graph. The names are chosen to be short to make it less dense on the screen. 'tsdy' is a special form of `regtail()` which does some extra analysis.

```
 >foo<             1    reg
                          brnc
                            piec
                              atom
 ><                4      tsdy~ EXACT <foo> (EXACT) (1)
                            ~ attach to END (3) offset to 2
```

The resulting program then looks like:

```
   1: EXACT <foo>(3)
   3: END(0)
```

As you can see, even though we parsed out a branch and a piece, it was ultimately only an atom. The final program shows us how things work. We have an EXACT regop, followed by an END regop. The number in parens indicates where the `regnext` of the node goes. The `regnext` of an END regop is unused, as END regops mean we have successfully matched. The number on the left indicates the position of the regop in the regnode array.

Now let's try a harder pattern. We will add a quantifier, so now we have the pattern `/foo+/`. We will see that `regbranch()` calls `regpiece()` twice.

```
 >foo+<            1    reg
                          brnc
                            piec
                              atom
 >o+<              3         piec
                              atom
 ><                6         tail~ EXACT <fo> (1)
                   7       tsdy~ EXACT <fo> (EXACT) (1)
                             ~ PLUS (END) (3)
                             ~ attach to END (6) offset to 3
```

And we end up with the program:

```
   1: EXACT <fo>(3)
   3: PLUS(6)
   4:   EXACT <o>(0)
   6: END(0)
```

Now we have a special case. The EXACT regop has a `regnext` of 0. This is because if it matches it should try to match itself again. The PLUS regop handles the actual failure of the EXACT regop and acts appropriately (going to regnode 6 if the EXACT matched at least once, or failing if it didn't).

Now for something much more complex: `/x(?:foo*|b[a][rR])(foo|bar)$/`

```
 >x(?:foo*|b...    1    reg
                          brnc
                            piec
                              atom
 >(?:foo*|b[...    3         piec
                              atom
 >?:foo*|b[a...                 reg
```

```
>foo*|b[a][...                  brnc
                                  piec
                                    atom
>o*|b[a][rR...    5               piec
                                    atom
>|b[a][rR])...    8               tail~ EXACT <fo> (3)
>b[a][rR])(...    9             brnc
                 10               piec
                                    atom
>[a][rR])(f...   12               piec
                                    atom
>a][rR])(fo...                        clas
>[rR])(foo|...   14               tail~ EXACT <b> (10)
                                  piec
                                    atom
>rR])(foo|b...                        clas
>)(foo|bar)...   25               tail~ EXACT <a> (12)
                                tail~ BRANCH (3)
                 26             tsdy~ BRANCH (END) (9)
                                    ~ attach to TAIL (25) offset to 16
                                tsdy~ EXACT <fo> (EXACT) (4)
                                    ~ STAR (END) (6)
                                    ~ attach to TAIL (25) offset to 19
                                tsdy~ EXACT <b> (EXACT) (10)
                                    ~ EXACT <a> (EXACT) (12)
                                    ~ ANYOF[Rr] (END) (14)
                                    ~ attach to TAIL (25) offset to 11
>(foo|bar)$<              tail~ EXACT <x> (1)
                          piec
                            atom
>foo|bar)$<                   reg
                 28             brnc
                                  piec
                                    atom
>|bar)$<         31             tail~ OPEN1 (26)
>bar)$<                         brnc
                 32               piec
                                    atom
>)$<             34             tail~ BRANCH (28)
                 36             tsdy~ BRANCH (END) (31)
                                    ~ attach to CLOSE1 (34) offset to 3
                                tsdy~ EXACT <foo> (EXACT) (29)
                                    ~ attach to CLOSE1 (34) offset to 5
                                tsdy~ EXACT <bar> (EXACT) (32)
                                    ~ attach to CLOSE1 (34) offset to 2
>$<                       tail~ BRANCH (3)
                             ~ BRANCH (9)
                             ~ TAIL (25)
                          piec
                            atom
><               37       tail~ OPEN1 (26)
                             ~ BRANCH (28)
                             ~ BRANCH (31)
                             ~ CLOSE1 (34)
                 38     tsdy~ EXACT <x> (EXACT) (1)
                             ~ BRANCH (END) (3)
```

```
                                ~ BRANCH (END) (9)
                                ~ TAIL (END) (25)
                                ~ OPEN1 (END) (26)
                                ~ BRANCH (END) (28)
                                ~ BRANCH (END) (31)
                                ~ CLOSE1 (END) (34)
                                ~ EOL (END) (36)
                                ~ attach to END (37) offset to 1
```

Resulting in the program

```
 1: EXACT <x>(3)
 3: BRANCH(9)
 4:   EXACT <fo>(6)
 6:   STAR(26)
 7:     EXACT <o>(0)
 9: BRANCH(25)
10:   EXACT <ba>(14)
12:   OPTIMIZED (2 nodes)
14:   ANYOF[Rr](26)
25: TAIL(26)
26: OPEN1(28)
28:   TRIE-EXACT(34)
      [StS:1 Wds:2 Cs:6 Uq:5 #Sts:7 Mn:3 Mx:3 Stcls:bf]
        <foo>
        <bar>
30:   OPTIMIZED (4 nodes)
34: CLOSE1(36)
36: EOL(37)
37: END(0)
```

Here we can see a much more complex program, with various optimisations in play. At regnode 10 we see an example where a character class with only one character in it was turned into an `EXACT` node. We can also see where an entire alternation was turned into a `TRIE-EXACT` node. As a consequence, some of the regnodes have been marked as optimised away. We can see that the `$` symbol has been converted into an `EOL` regop, a special piece of code that looks for `\n` or the end of the string.

The next pointer for `BRANCH`es is interesting in that it points at where execution should go if the branch fails. When executing if the engine tries to traverse from a branch to a `regnext` that isn't a branch then the engine will know that the entire set of branches have failed.

Peep-hole Optimisation and Analysis

The regular expression engine can be a weighty tool to wield. On long strings and complex patterns it can end up having to do a lot of work to find a match, and even more to decide that no match is possible. Consider a situation like the following pattern.

```
'abababababababababababab' =~ /(a|b)*z/
```

The `(a|b)*` part can match at every char in the string, and then fail every time because there is no `z` in the string. So obviously we can avoid using the regex engine unless there is a `z` in the string. Likewise in a pattern like:

```
/foo(\w+)bar/
```

In this case we know that the string must contain a `foo` which must be followed by `bar`. We can use

Fast Boyer-Moore matching as implemented in `fbm_instr()` to find the location of these strings. If they don't exist then we don't need to resort to the much more expensive regex engine. Even better, if they do exist then we can use their positions to reduce the search space that the regex engine needs to cover to determine if the entire pattern matches.

There are various aspects of the pattern that can be used to facilitate optimisations along these lines:

* anchored fixed strings

* floating fixed strings

* minimum and maximum length requirements

* start class

* Beginning/End of line positions

Another form of optimisation that can occur is post-parse "peep-hole" optimisations, where inefficient constructs are replaced by more efficient constructs. An example of this are `TAIL` regops which are used during parsing to mark the end of branches and the end of groups. These regops are used as place-holders during construction and "always match" so they can be "optimised away" by making the things that point to the `TAIL` point to thing that the `TAIL` points to, thus "skipping" the node.

Another optimisation that can occur is that of "`EXACT` merging" which is where two consecutive `EXACT` nodes are merged into a single regop. An even more aggressive form of this is that a branch sequence of the form `EXACT BRANCH ... EXACT` can be converted into a `TRIE-EXACT` regop.

All of this occurs in the routine `study_chunk()` which uses a special structure `scan_data_t` to store the analysis that it has performed, and does the "peep-hole" optimisations as it goes.

The code involved in `study_chunk()` is extremely cryptic. Be careful. :-)

Execution

Execution of a regex generally involves two phases, the first being finding the start point in the string where we should match from, and the second being running the regop interpreter.

If we can tell that there is no valid start point then we don't bother running interpreter at all. Likewise, if we know from the analysis phase that we cannot detect a short-cut to the start position, we go straight to the interpreter.

The two entry points are `re_intuit_start()` and `pregexec()`. These routines have a somewhat incestuous relationship with overlap between their functions, and `pregexec()` may even call `re_intuit_start()` on its own. Nevertheless other parts of the the perl source code may call into either, or both.

Execution of the interpreter itself used to be recursive. Due to the efforts of Dave Mitchell in the 5.9.x development track, it is now iterative. Now an internal stack is maintained on the heap and the routine is fully iterative. This can make it tricky as the code is quite conservative about what state it stores, with the result that that two consecutive lines in the code can actually be running in totally different contexts due to the simulated recursion.

Start position and no-match optimisations

`re_intuit_start()` is responsible for handling start points and no-match optimisations as determined by the results of the analysis done by `study_chunk()` (and described in *Peep-hole Optimisation and Analysis*).

The basic structure of this routine is to try to find the start- and/or end-points of where the pattern could match, and to ensure that the string is long enough to match the pattern. It tries to use more efficient methods over less efficient methods and may involve considerable cross-checking of constraints to find the place in the string that matches. For instance it may try to determine that a given fixed string must be not only present but a certain number of chars before the end of the string, or whatever.

It calls several other routines, such as `fbm_instr()` which does Fast Boyer Moore matching and `find_byclass()` which is responsible for finding the start using the first mandatory regop in the program.

When the optimisation criteria have been satisfied, `reg_try()` is called to perform the match.

Program execution

`pregexec()` is the main entry point for running a regex. It contains support for initialising the regex interpreter's state, running `re_intuit_start()` if needed, and running the interpreter on the string from various start positions as needed. When it is necessary to use the regex interpreter `pregexec()` calls `regtry()`.

`regtry()` is the entry point into the regex interpreter. It expects as arguments a pointer to a `regmatch_info` structure and a pointer to a string. It returns an integer 1 for success and a 0 for failure. It is basically a set-up wrapper around `regmatch()`.

`regmatch` is the main "recursive loop" of the interpreter. It is basically a giant switch statement that implements a state machine, where the possible states are the regops themselves, plus a number of additional intermediate and failure states. A few of the states are implemented as subroutines but the bulk are inline code.

MISCELLANEOUS

Unicode and Localisation Support

When dealing with strings containing characters that cannot be represented using an eight-bit character set, perl uses an internal representation that is a permissive version of Unicode's UTF-8 encoding[2]. This uses single bytes to represent characters from the ASCII character set, and sequences of two or more bytes for all other characters. (See *perlunitut* for more information about the relationship between UTF-8 and perl's encoding, utf8 -- the difference isn't important for this discussion.)

No matter how you look at it, Unicode support is going to be a pain in a regex engine. Tricks that might be fine when you have 256 possible characters often won't scale to handle the size of the UTF-8 character set. Things you can take for granted with ASCII may not be true with Unicode. For instance, in ASCII, it is safe to assume that `sizeof(char1) == sizeof(char2)`, but in UTF-8 it isn't. Unicode case folding is vastly more complex than the simple rules of ASCII, and even when not using Unicode but only localised single byte encodings, things can get tricky (for example, **LATIN SMALL LETTER SHARP S** (U+00DF, ß) should match 'SS' in localised case-insensitive matching).

Making things worse is that UTF-8 support was a later addition to the regex engine (as it was to perl) and this necessarily made things a lot more complicated. Obviously it is easier to design a regex engine with Unicode support in mind from the beginning than it is to retrofit it to one that wasn't.

Nearly all regops that involve looking at the input string have two cases, one for UTF-8, and one not. In fact, it's often more complex than that, as the pattern may be UTF-8 as well.

Care must be taken when making changes to make sure that you handle UTF-8 properly, both at compile time and at execution time, including when the string and pattern are mismatched.

The following comment in *regcomp.h* gives an example of exactly how tricky this can be:

```
Two problematic code points in Unicode casefolding of EXACT nodes:

U+0390 - GREEK SMALL LETTER IOTA WITH DIALYTIKA AND TONOS
U+03B0 - GREEK SMALL LETTER UPSILON WITH DIALYTIKA AND TONOS

which casefold to

Unicode                      UTF-8
```

```
    U+03B9 U+0308 U+0301         0xCE 0xB9 0xCC 0x88 0xCC 0x81
    U+03C5 U+0308 U+0301         0xCF 0x85 0xCC 0x88 0xCC 0x81

    This means that in case-insensitive matching (or "loose matching",
    as Unicode calls it), an EXACTF of length six (the UTF-8 encoded
    byte length of the above casefolded versions) can match a target
    string of length two (the byte length of UTF-8 encoded U+0390 or
    U+03B0). This would rather mess up the minimum length computation.

    What we'll do is to look for the tail four bytes, and then peek
    at the preceding two bytes to see whether we need to decrease
    the minimum length by four (six minus two).

    Thanks to the design of UTF-8, there cannot be false matches:
    A sequence of valid UTF-8 bytes cannot be a subsequence of
    another valid sequence of UTF-8 bytes.
```

Base Struct

regexp.h contains the base structure definition:

```
    typedef struct regexp {
            I32 *startp;
            I32 *endp;
            regnode *regstclass;
            struct reg_substr_data *substrs;
            char *precomp;          /* pre-compilation regular expression
*/
            struct reg_data *data;  /* Additional data. */
            char *subbeg;           /* saved or original string
                                       so \digit works forever. */
            U32 *offsets;           /* offset annotations 20001228 MJD */
            I32 sublen;             /* Length of string pointed by subbeg
*/
            I32 refcnt;
            I32 minlen;             /* minimum possible length of $& */
            I32 prelen;             /* length of precomp */
            U32 nparens;            /* number of parentheses */
            U32 lastparen;          /* last paren matched */
            U32 lastcloseparen;     /* last paren matched */
            U32 reganch;            /* Internal use only +
                                       Tainted information used by regexec?
 */
            regnode program[1];     /* Unwarranted chumminess with
compiler. */
    } regexp;
```

`program`, and `data` are the primary fields of concern in terms of program structure. `program` is the actual array of nodes, and `data` is an array of "whatever", with each whatever being typed by letter, and freed or cloned as needed based on this type. regops use the data array to store reference data that isn't convenient to store in the regop itself. It also means memory management code doesn't need to traverse the program to find pointers. So for instance, if a regop needs a pointer, the normal procedure is use a `regnode_arg1` store the data index in the `ARG` field and look it up from the data array.

-

`startp`, `endp`, `nparens`, `lasparen`, and `lastcloseparen` are used to manage capture buffers.

- `subbeg` and optional `saved_copy` are used during the execution phase for managing replacements.

- `offsets` and `precomp` are used for debugging purposes.

- The rest are used for start point optimisations.

De-allocation and Cloning

Any patch that adds data items to the regexp will need to include changes to *sv.c* (`Perl_re_dup()`) and *regcomp.c* (`pregfree()`). This involves freeing or cloning items in the regexes data array based on the data item's type.

SEE ALSO

perlre

perlunitut

AUTHOR

by Yves Orton, 2006.

With excerpts from Perl, and contributions and suggestions from Ronald J. Kimball, Dave Mitchell, Dominic Dunlop, Mark Jason Dominus, Stephen McCamant, and David Landgren.

LICENCE

Same terms as Perl.

REFERENCES

[1] *http://perl.plover.com/Rx/paper/*

[2] *http://www.unicode.org*