PFC

Functional Parser Combinators in Perl

Paul Anderson <dcspaul@nine3.org>
nine3.org

1 PFC

PFC is a collection of *functional parser combinators* inspired by Parsec[1]. It also includes a *generator* to generate combinator code from a simple EBNF grammar, and an *explorer* to render an AST in interactive HTML for exploring the parse tree.

1.1 Parser Functions

PFC functions return parser functions. For example:

```
# a (double) quoted string
my $f = PFCString;
# a keyword
# (terminated by a word boundary)
my $g = PFCKeyWord('begin');
```

The resulting parser functions (`$f`, `$g`) accept a `Pos` object which includes the source string and an optional pathname (for error reporting only)[2]. The second argument is a `PFC::Expect` object which keeps track of alternative parses:

```
my $p = new PFC::Pos($src,$path);
my $isok = &$f($p,new PFC::Expect);
```

If there is a fatal error - e.g. a filesystem error - the parser function returns `false` and `$@` contains a structured error message (see 1.2) which can be rendered as a string with the `Fmt()` function:

```
die Fmt($@) unless &$f(...);
```

If the parser function returns `true`, `$@` contains either a `Token` object (if the parse succeeds), or an `Expect` object (if the parse fails). The `Token` object represents the resulting parse tree, including the value and source location for each node. The `Expect` object contains details of the alternative input symbols which would have allowed the parse to proceed.

The method `Succeeded` returns `true` for a `Token` and `false` for an `Expect`, so a parser can be invoked on a source string `$str` as follows:

```
my $p = new PFC::Pos($src,$path);
die Fmt($@)
  unless &$f($p,new PFC::Expect);
die $@->AsString
  unless ($@->Succeeded);
my $token = $@;
```

The `Value` of the resulting token depends on the particular PFC function. For terminals such as `$f` and `$g` above, the value is the string value of the terminal. For combinators (see below), the value is usually a list of others tokens.

The `Pos` and `End` methods of the token return the starting and ending position of the token in the source stream. These can be rendered as strings with the `AsString()` method.

1.2 Errors

The internal representation of an error is not always a simple string: it may be an object, or a (recursive) list of errors. This structured format makes it easier to identify the error locations should we ever need them - for example in a GUI interface. If you need a simple string, apply the `Fmt` function to the error.

1.3 Combinators

PFC *combinators* are functions which take (references to) other parser functions and return a function which parses some combination of them. For example, `PFCSeq` parses a sequence, so we could parse an identifier followed by a string with:

```
my $f = PFCSeq( \&PFCID, \&PFCString );
```

Other combinators (see 4) parse alternative choices (`PFCAlt`), optional items (`PFCOPt`), lists (`PFCList`) and sequences of items separated by a given separator (`PFCSepBy`). The combinators which parse sequences and lists of items return lists as their value.

If the PFC functions require an argument, then the Perl syntax becomes a little clumsy:

```
my $f = PFCSeq(
  sub { PFCKeyWord('print'); },
  \&PFCString );
```

The PFC generator (see 2) can generate this Perl code from a simple EBNF specification and this is recommended for any non-trivial application.

[1] https://caiorss.github.io/Functional-Programming/haskell/Parsec_parser_combinators.html

[2] The `Pos` constructor also accepts some additional optional arguments to specify an offset within the string

1.4 WhiteSpace

The `PFCLexeme` combinator takes any parser function and returns a *lexeme* version - i.e. a parser which skips any leading white space. However, all of the standard PFC functions except `PFCRegEx` are lexeme by default.

1.5 Transforming Node Values

The default value returned by a parser is a tree with the terminal nodes representing the text of individual tokens, and the non-terminal nodes being lists of other nodes (generated by combinators).

Sometimes it is useful to transform these nodes as they are parsed, rather than delaying this until the end of the parse. For example, a string and an identifier would both appear as a simple terminal node in the parse tree, but we would usually want to treat these differently. The `PFCApply` function allows an arbitrary function to transform the value of the node when it is recognised.

For example, the following function is a parser which returns the upper case value of an identifier:

```
sub MakeUpper($$$) {
  my ($v,$pos,$end) = @_;
  return OK(uc($v));
}
my $f = PFCApply(\&MakeUpper,\&PFCID);
```

The start and end positions of the token are available to the function but cannot be changed.

In practice, such functions would typically return different objects representing the different types of token. However, *these functions must not change any global state*, since the parser may backtrack over the token.

1.6 Handling Newlines and EOF

Newlines are normally treated as white-space and will be skipped by the lexeme parsers. The function `PFCEOL` succeeds only at the end of a line (and skips to the start of the next line). `PFCBOL` succeeds only if the next non-space character is at the beginning of a line. `PFCEOF` succeeds only at the end of a file.

When parsing line-oriented languages, it is useful if the line endings are not treated as white space. The `PFCOneLine` combinator returns a parser which calls its argument function with newlines not treated as white-space.

1.7 Including Files

The `PFCInclude` function supports the inclusion of other source files. The argument to `PFCInclude` should be a parser function which returns a token with a string value that will be interpreted as the pathname of the file to include. For example:

```
sub Include {
  PFCInclude(
    PFCApply(
      sub { OK($_[0]->[1]->Value); },
      \&IncFile
    )
  )
}
sub IncFile {
  PFCSeq(
    sub { PFCKeyWord("#include"); }
    \&Literal
    sub { PFCSym(";"); }
  );
}
```

Notice:

- The rule which parses the `#include` statement (`IncFile`) must parse the entire statement - including any terminator. Otherwise the terminator will appear in the input stream *following* the content of included file.
- An additional function is required (the argument to the `PFCApply` above) to to extract the pathname from the parse tree of the include statement and return it as a single string token.
- The name of the include file itself will appear in the AST as a node immediately before the included content.

The generator (see below) provides a simple way of generating the code for include statements.

1.8 Lookahead

PFC grammars are LL(1)[3], but two combinators are provided which can "look ahead" to manually handle productions which cannot be distinguished by just looking at the next symbol:

```
PFCFollowedBy(\&f,\&g)
```

returns the result of parsing `f()`, but only if `g()` succeeds on the following string. Similarly

```
PFCNotFollowedBy(\&f,\&g)
```

returns the result of `f()` only if `g()` fails on the following string. Note that PFC can not generate a sensible "expectation" message if this combinator fails (because of the negation), so this combinator should al-

[3] https://www.garshol.priv.no/download/text/bnf.html

ways have a custom error message applied (see below).

Simple lookahead can also be handled using regular expressions. For example, a digit not followed by an equals sign:

```
Value := /[0-9]*(?!(\s*=))/
```

1.9 Custom Error Messages

The `Expect` object returned from a parse fail normally lists all of the tokens which would have been acceptable at the point of failure. Sometimes, this can be confusingly long. The `PFCDescr` combinator returns a function which applies the argument parser but returns the specified message if it fails. This allows a message such as "`expected a '+' or a '-' or a '*' or a '/'`" to be replaced with "`expected an arithmetic operator`".

2 The Generator

The Perl syntax can make the parser expressions unwieldy for any non-trivial language. The PFC generator takes a language description in an extended BNF format and generates Perl code for the parser.

The following command will locate a `.pfc` file matching the given *regex* (assumed to contain eBNF) in the current project hierarchy (under `$::topdir`), and create a corresponding `.pm` file containing the Perl code.

```
ptools pfc gen regex
```

The generator parses the source EBNF with a parser generated by itself. The file `Grammar.pfc` in the distribution is the source language for the generator.

2.1 Production Rules

The generator input consists of list of production rules of the form:

```
name := expression
```

Each of these creates a Perl parser function with the same name. These can be freely mixed with manually-defined parser functions.

User-defined function names may need to be fully qualified, since the generated output appears in its own package.

2.2 Terminals

The following terminal expressions are supported:

- *identifier* - a rule name or other PFC function (with no arguments)
- "*string*" - a literal value
- '*string*' - a keyword (ending on a word boundary)
- /*regex*/ - a regular expression

The regular expression may be followed by any of the usual Perl flags as well as `/L` to create a lexeme parser (the default `PFCRegEx` is not lexeme).

2.3 Combinators

The standard syntax is supported for:

- sequences: `A B ...`
- lists `A*`
- non-empty lists `A+`
- alternatives `A|B|...`
- optional values `A?`

`PFCSepBy` is also supported as: `A ! SEP *`

A minimum number of repeats can be specified by following the `*` with an integer, and brackets (`()`) can be used in expressions to group the combinators.

Lookahead is supported as:

- `A~B` matches `A`, but only when followed by `B`.
- `A!~B` matches `A`, but only when not followed by `B`.

2.4 Additional Features

`PFCOneLine` is supported by enclosing an expression in braces (`{}`).

`PFC Descr` is supported by following an expression with text in angle brackets (<>).

Single-line comments are supported using `//`.

2.5 Function application

If a Perl package is specified with the `#use` directive, then the generated grammar will automatically include `PFCApply` for every *rule* where the function `PFC_`*rule* appears in the package. The default prefix (`PFC_`) can be changed with the `#prefix` directive (this affects only rules which follow the directive).

```
#use MyApp::MyPackage
#prefix MyFun_
```

2.6 Including Files

The generator provides a convenient way of creating the code to support file inclusion. For example, the code in section 1.7 can be generated with the following

rules:

```
Include := @ IncFile 1
IncFile := '#include' Literal ";"
```

The `@`*name* expression parses the named rule and uses the return value as the name of the source file to be included. By default `IncFile` would require an extra function application to extract the pathname from the returned AST. However, if the named rule returns a simple sequence (as above), then it can be followed by a numeric index value and the corresponding element will be extracted from the sequence without the need for an additional function.

Notice that the statement terminator must appear in the `IncFile` and not in the `Include`, otherwise the terminator will appear in the input stream *following* the contents of the included file.

The generator itself uses this mechanism to implement the `#import` directive which can be used to include other files into the grammar specification.

2.7 Latex Output

The generator can also render the grammar in Latex suitable for inclusion in documentation:

```
ptools pfc latex regex -o outputfile.tex
```

The `#section` directive can be used to split large grammars into named sections[4].

```
#section section heading
```

See appendix B for the generator grammar itself rendered in Latex.

3 The Explorer

The `pfc` tool can generate an HTML display which allows the parse tree to be explored by folding/unfolding the rules and highlighting the corresponding source section:

```
ptools pfc ex regex node@path
```

See appendix A for example output.

The *regex* should match the name of the required grammar file, and the corresponding `.pm` file should have previously been generated using `ptools pfc gen`.

The *path* is the pathname of the file containing the source to be explored, and the *node* is the name of the AST node which represents the top-level of the source file.

The `-R` option can be used to inhibit the application of the `PFCApply` functions. This allows the raw parse tree to be explored, rather than the version which has been processed by the function applications. Functions can be named `PFC_Force`*name* instead of simply `PFC_`*name* to force them to be applied even when the -R option is specified. This is necessary for example for functions which handle inclusion/importing of other files.

Using the argument `ast` instead of `ex` will produce text output of the AST. The `-o` option can be used to place the output in a specified file.

4 Function Reference

PFCAlt(*f*,...): Return a function to parse one of the alternatives *f*.... The first matching alternative is returned. Note that evaluation stops after the first match. This means that (a) elements following the match will not be recorded as possible "expectations", and (b) there is no backtracking so the first match is a committed choice.

PFCAngleString: Return a function to parse a string enclosed in angle brackets (<>). Return the content of the string.

PFCApply(*t*,*f*): Return a function to call the parser function *f* and transform the resulting value with the function *t*.

PFCBOL: Return a parser function which succeeds at the beginning of as line. The function returns `undef`.

PFCDescr(*descr*,*f*): Return a parser function which calls the parser function *f* and returns *descr* as the expectation if *f* fails. Otherwise, return the return value of *f*.

PFCEOF: Return a parser function which succeeds at the end of the source file. The function returns `undef`.

PFCEOL: Return a parser function which succeeds at the end of a line. The function returns `undef`.

PFCFollowedBy(*f*,*g*): Return a parser function which succeeds if `f` succeeds, but only if `g` succeeds on the following text (the input parsed by `g` is not consumed).

PFCHID: Return a parser function which parses an hyphenated identifier. The function returns the name of the identifier (string).

PFCID: Return a parser function which parses an identifier. The function returns the name of the identifier (string).

[4] This directive is ignored when generting code for the grammar.

PFCInclude(*f*): Call the parser function *f* to obtain the pathname of a file. The function returns the result of parsing the contents of the file.

PFCKeyWord(*k*): Return a parser function which parses the text *k*. The text of the keyword must be followed by a word boundary - see also `PFCSym`. The function returns the keyword (string).

PFCLexeme(*f*): Return a parser function which skips white-space before calling the parser function *f*.

PFCList(*f*): Return a parser function which parses a list of tokens parsed by the function *f*. Note that this is greedy and committed, so (for example) `X*X` will never succeed.

PFCNotFollowedBy(*f*,*g*): Return a parser function which succeeds if `f` succeeds, but only if `g` does not succeed on the following text (the input parsed by `g` is not consumed).

PFCOneLine(*f*): Return a parser function which calls the function *f* while treating newlines as non-space.

PFCOpt(*f*): Return a parser function which calls the function *f* and succeeds (returning `undef`) if *f* fails.

PFCRegEx(*re*): Return a function to parse a regular expression enclosed in slashes (`/`). The function returns the match of the regular expression.

PFCREString: Return a function to parse a string enclosed in slashes (`/`). The function returns the content of the string.

PFCSepBy(*sep*,*f*): Return a parser function which parses a list of tokens parsed by the function *f*, separated by tokens parsed by the function *sep*. Note that this is greedy and committed, so (for example): `X!Y*X` will never succeed.

PFCSeq(*f*,...): Return a parser function which parses a sequence of tokens parsed by *f*....

PFCSingleString: Return a function to parse a string enclosed in single quotes (`''`). The function returns the content of the string.

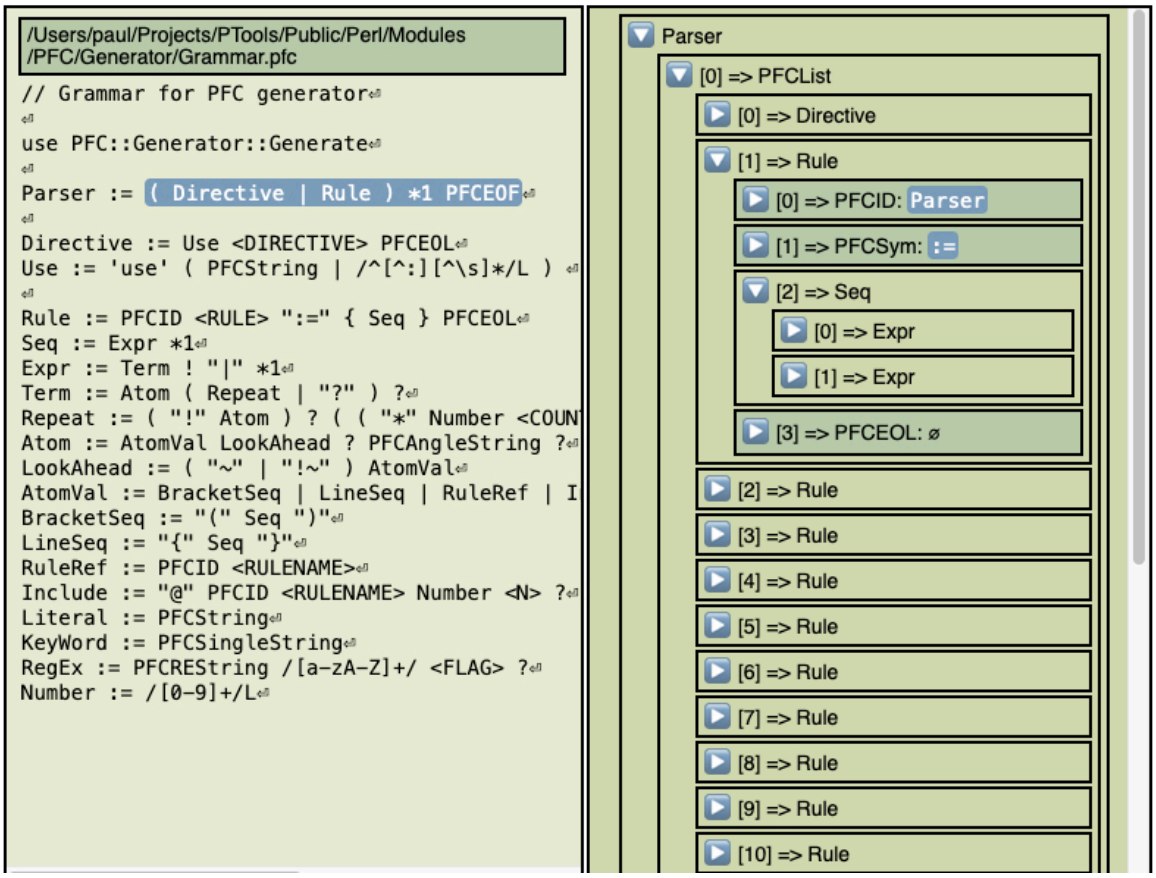
PFCSquareString: Return a function to parse a string enclosed in square brackets (`[]`). Return the content of the string.

PFCString: Return a function to parse a string enclosed in double quotes (`""`). The function returns the content of the string.

PFCSym(*s*): Return a function to parse a literal symbol (text). The function returns the text string. Note that *s* may not be followed by a word boundary - (see also `PFCKeyWord`).

PFCType(*t*,*f*): Return a parser function which calls the function *f* and attaches the string *t* to the resulting token as the *type* (if the function succeeds). The generator uses this to add the rule names as the types. This is then shown by the explorer when viewing the AST node, and can be used to distinguish between (for example) different types of node which both have strings as their values.

Appendix A Explorer Interface Example



Appendix B Generator Grammar

Parser := (Directive | Rule) + *EOF*

Directives

Directive := (Use | ImportDirective | Prefix | Section)

ImportDirective := ImportArg[(a)]

Use := `#use` DirectiveArg

Prefix := `#prefix` DirectiveArg

ImportArg := `#import` DirectiveArg

Section := `#section` DirectiveArg

DirectiveArg := (`"` ... `"` | `/^[^:][^\s]*/`[(d)]) *EOL*

Rules

Rule := *IDENTIFIER* `:=` (Seq)[(b)] *EOL*

Seq := Expr +

Expr := Term (`|` Term) *

Term := Atom (Repeat | `?`) ?

Repeat := (`!` Atom) ? ((`*` Number ?) | `+`)

Atom := AtomVal LookAhead ? `<` ... `>` ?

LookAhead := (`~` | `!~`) AtomVal

AtomVal := BracketSeq | LineSeq | RuleRef | Include | Literal | KeyWord | RegEx

BracketSeq := `(` Seq `)`

LineSeq := `{` Seq `}`

RuleRef := *IDENTIFIER*

Include := `@` *IDENTIFIER* Number ?

Literal := `"` ... `"`

KeyWord := `'` ... `'`

RegEx := `/` ... `/` `/[a-zA-Z]+/` ?

Number := `/[0-9]+/`[(d)]

(a) The value of this rule is interpreted as a filename which is expected to include additional grammar rules.

(b) Newlines are not treated as whitespace during evaluation of this term – i.e. the term must appear on one line, unless explicit continuation lines are given.

(d) Leading whitespace is skipped before matching this regex.