# Advanced Perl Parsing

**Damian Conway**

**School of Computer Science and Software Engineering**
**Monash University**
**Australia**

**damian@conway.org**

**Deutscher Perl-Workshop 3.0**
**28 February 2001**

# Tutorial Summary

## Session 1

| Topic | Examples | Concepts |
|---|---|---|
| **Text::Balanced** *Page 4* | *#para..#endpara* | Tagged text, multiple extraction |
| **Miscellanea** *Page14* | *Paragraph parsing*<br>*Bad version numbers* | Named items, parsing context, rejection |
| **Debugging grammars** *Page 19* | | Debugging parser construction, debugging parser execution |
| **Ambiguous input** *Page 25* | *Comma/colon/tab-separated values*<br>*NL sentences* | Non-deterministic parsing |
| **Automation** *Page 32* | *Incremental development*<br>*Infix to RPN* | Autostubbing, autoactions, autotrees, autogenerators |
| **Generic parsing** *Page 44* | *Block structured languages*<br>*Finding DNA sequences* | Run-time subrule selection, parametric rules |

# Tutorial Summary

## Session 2

| Topic | Examples | Concepts |
|---|---|---|
| **Distributed text**<br>*Page 53* | *A Lisp-like language* | Lookahead for multiparsing, recursive file inclusion, concatenative file inclusion |
| **Self-modifying grammars**<br>*Page 63* | *Parsing C/C++ declarations*<br>*Category hierarchies* | Runtime rule replacement and extension, grammatical learning |
| **(Nearly) Parsing Perl**<br>*Page 75* | *CSPQV parsing*<br>*Ors2oars*<br>*The winnow function* | Source code filtering, code winnowing |
| **Metagrammars**<br>*Page 88* | *Mission statements*<br>*Beat poetry*<br>*Postmodernist literature* | Grammars that parse grammars, text generation |

# Resources

- `http//www.perl.com/CPAN/authors/id/DCONWAY/`

- `http//www.csse.monash.edu.au/~damian/TPC/Tutorial/AdvParsing`

- Aho, Sethi & Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1985. ("The dragon book")

- Pittman & Peters, *The Art of Compiler Design*, Prentice Hall, 1992.

- Levine, Mason & Brown, *lex & yacc*, 2nd Edition, O'Reilly, 1992.

- Conway, *The `man(1)` of Descent*, in "The Perl Journal", Issue #12, pp. 46-58, 1998.

- The Parse::RecDescent module POD manual

# Text::Balanced revisited

- The Text::Balanced module has many powerful parsing features beyond basic delimited-string and balanced-brackets matching.

- It can parse Perl variables and entire blocks of Perl code.

- It can parse tagged mark-up.

- It can do all these things at once.

# Extracting tagged text

- A very common parsing task is to extract text between delimiters that are more complex than a single character (i.e. tags).

- Tags often come in pairs, but not always.

- XML:
  ```
  <tag> ... </tag>
  <tag/>
  ```

- HTML:
  ```
  <P> ... </P>
  <P> ...
  ```

- POD sections:
  ```
  =begin ... =end
  =for ...
  ```

- POD markup:
  ```
  B< ... >
  ```

- Sometimes the intervening text is allowed to contain nested tags (e.g. POD markup); sometimes it's not (e.g. POD directives); sometimes it depends (e.g. HTML <B>s nest, but <A>'s don't)

- Sometimes the trailing tag is optional. In such cases the “contained” text may be defined to be everything up to the next identical tag (e.g. <P> in HTML), or just the next paragraph (e.g. `=for` in POD)

- All in all, it's a nightmare.

- If you're lucky, it will already have been someone else's nightmare and there will be a CPAN module to help: Pod::Parser, HTML::Parser, XML::Parser.

- But (as yet) there's no The::Godawful::Braindead::Markup::Notation::My::Unprintable::Boss::Insists::We::Use::Parser in the CPAN.

- That's were Text::Balanced's `extract_tagged` subroutine comes in handy.

- `extract_tagged` takes a string, plus four arguments specifying the type of tag being parsed, and tries to extract suitably tagged text from the start of the string.

- For example, consider a markup language where the paragraph tags are `#para` and `#endpara` respectively. The following code would extract such a paragraph (after skipping any leading whitespace):

```
use Text::Balanced "extract_tagged";

($extracted,$remainder,$prefix,$starttag,$text,$endtag)
   = extract_tagged($text, "#para", "#endpara");
```

- By default, `extract_tagged` allows tags to nest, but if that is not the desired behaviour, you can specify which tags may not appear in the text:

```
($extracted,$remainder,$prefix,$starttag,$text,$endtag)
   = extract_tagged($text, "#para", "#endpara", undef,
                    { reject => ["#para"] } );
```

- Of course, if the #endpara tag were optional, then encountering another #para tag should merely indicate the end of the current paragraph:

```
($extracted,$remainder,$prefix,$starttag,$text,$endtag)
  = extract_tagged($text, "#para", "#endpara", undef,
                   { reject => ["#para"],
                     fail   => "MAX"   });
```

- Alternatively, unbalanced #para tags may only apply to the next chunk of text (up to the next empty line). extract_tagged can handle that too:

```
($extracted,$remainder,$prefix,$starttag,$text,$endtag)
  = extract_tagged($text, "#para", "#endpara", undef,
                   { reject => ["#para"],
                     fail   => "PARA"   });
```

- The tag specifiers are actually interpolated into patterns, so you could handle multiple tags (#para ... #endpara, #table ... #endtable, #anger ... #endanger, etc.) with a single call:

```
do {
($extracted,$remainder,$prefix,$starttag,$text,$endtag)
   = extract_tagged($text, '#([a-z]+)', '#end$1')
} while $extracted;
```

- Note the use of uninterpolated quotes for the tags.

# Extracting mixed components

- Real data is rarely as homogeneous as the examples shown above.

- Typically a data file will have a variety of components: keywords, operators, delimited data, tagged sections, etc:

```
name = Damian
rank = "minor demi-urge"
"sig key":
  alpha,+bravo charlie+,delta,+echo foxtrot+

{NOTE}
    This file format needs some serious redesigning!
{/NOTE}

EODATA
```

- So you will often find yourself writing something like this:

```
$_ = join "", <>;

my @data;
sub add_data
{
        my ($type, $match) = @_;
        push @data, bless \$match, $type;
}

my $single_quote = gen_delimited_pat(q{'});
my $double_quote = gen_delimited_pat(q{"});
my $delimited    = gen_delimited_pat(q{+},q{+});
my $comment      = gen_extract_tagged("{NOTE}");

while (pos() < length())
{
        if    (/\G\s*([a-z]+) = /gc)
                        { add_data(Identifier => $1) }

        elsif (/\G\s*($single_quote)/gco)
                        { add_data(String     => $1) }

        elsif (/\G\s*($double_quote):/gco)
                        { add_data(Identifier => $1) }

        elsif (/\G\s*,?($delimited)/gco)
                        { add_data(String     => $1) }

        elsif ($comment->($_))
                        { next }

        elsif (/\G\s*(EODATA)/gc)
                        { add_data(EOF        => $1) }

        elsif (/\G\s*,?([a-z]+)/gci)
                        { add_data(Bareword   => $1) }

        else            { /./gcs }
}
```

- This is equivalent to a really complicated `split`, except that `split` can't mix in calls to extraction functions.

- Text::Balanced provides a means of automating the task of breaking up text into fields in this way: `extract_multiple`.

- The previous example could be rewritten:

```
$_ = join "", <>;

my $single_quote = gen_delimited_pat(q{'});
my $double_quote = gen_delimited_pat(q{"});
my $delimited    = gen_delimited_pat(q{+},q{+});
my $comment      = gen_extract_tagged("{NOTE}");

my @data = extract_multiple($_,
                [
                  { Identifier => qr/\s*([a-z]+) = /      },
                  { String     => qr/\s*($single_quote)/  },
                  { Identifier => qr/\s*($double_quote):/ },
                  { String     => qr/\s*,?($delimited)/   },
                  sub { "" if $comment->(shift) },
                  { EOF        => '\s*(EODATA)'           },
                  { Bareword   => qr/\s*,?([a-z]+)/i      },
                ], undef, 1);
```

- The subroutine takes a string to be broken up, a list of field extractors, and two extra arguments (see below).

- It loops through the extractor set looking for an extractor that matches the next piece of the string.

- Extractors may be subroutines, or precompiled patterns, or simple strings.

- Subroutine extractors are expected to match against the string, set its `pos`, and return a list. If the list is empty, the extractor failed. Otherwise, the first item of the list is the extracted field.

- Pattern and string extractors are interpolated into a `m/.../gc` match. If the match is successful, `$1` (or `$&` if there is no `$1`) is the extracted field.

- They may also be specified as the value of a hash, whose key is the name of a class into which the resulting string fragment will be blessed.

- The third argument to `extract_multiple` is the maximal number of fields into which the string is to be split. Its purpose is the same as the third argument of a `split`.

- The final argument is a boolean value, telling the subroutine whether to ignore anything that's not matched by an extractor.

- If the last argument is false (or omitted), any text between extracted fields is also returned. For example, the following code separates Perl variables from the surrounding Perl code:

```
@vars_et_al = extract_multiple($source, [
                  sub { extract_variable $_[0] }
              ]);
```

- And here's yet another way to do CSV parsing:

```
@fields = extract_multiple($csv_text, [
              sub { extract_delimited($_[0],q{'"}) },
              qr/([^,]+)(.*)/,
          ], undef, 1);
```

# Parse::RecDescent miscellanea

## Named items

- As grammars evolve, the use of the `@item` array to access parsed elements within actions can lead to hard-to-track bugs.

- For example, consider a rule that starts its life as:

```
paragraph: bullet firstline otherlines
             { Para->new(-text => $item[2].$item[3],
                         -bullet => $item[1]);
             }
```

- Later, it may become important to track leading whitespace within the subrules (to assist in formatting), so the rule becomes:

```
paragraph: <skip:""> bullet firstline otherlines
             { Para->new(-text => $item[2].$item[3],
                         -bullet => $item[1]);
             }
```

- ...and the paragraphs suddenly get much shorter!

- To avoid this irritation, RecDescent provides an alternate access mechanism for parsed elements: the `%item` hash.

- The `%item` hash stores the same set of subrule results as the `@item` array, but the individual items are stored under *logical* keys, rather than *positional* indices.

- Rewriting the original rule to use `%item` produces:

```
paragraph: bullet firstline otherlines
             { Para->new(-text => $item{firstline} .
                                  $item{otherlines},
                         -bullet => $item{bullet});
             }
```

- Now, when the `<skip:"">` is prepended to the production, the action continues to work correctly without modification.

- This works well for named subrule items, but what about other components (i.e. literals, patterns, directives, or actions)?

- All of these items are also stored in the `%item` hash, but under a "semi-positional" key. For example:

```
paragraph: "*" /.*\n/ { read2eop ($text) }
             { Para->new(-text => $item{__PATTERN1__} .
                                  $item{__ACTION1__},
                         -bullet => $item{__STRING1__})
             }
```

- Note that these keys would also be unaffected by the insertion of a `<skip:"">` directive, although adding in another pattern, action, or literal string may still require one of the keys to be modified.

# Terminating with extreme prejudice

- Sometimes in the middle of a parse it's possible to determine that the grammar can (or should) never succeed.

- For example, suppose we were parsing a word-processor data file:

```
datafile:   preamble(s) layout(s)

preamble:   encoding | macros | version

version:    "vrsn:" /\d\d[.]\d\d/<reject: do{$item[2]>3.62} >
        |   <error: Cannot read files for versions after 3.62>

layout:     # etc.
```

- This generates the appropriate error message if it encounters a file format later than v03.62.

- That failure causes preamble processing to cease, but the parser still attempts to parse the layout section of the file.

- Instead of an <error> directive, we need something more terminal. We need the <fatal> directive.

- A <fatal> directive is just like an <error> directive, except that, instead of just causing the current rule to fail, it causes the entire parser to give up immediately.

- So we could rewrite the previous parser:

```
datafile:   preamble(s) layout(s)

preamble:   encoding | macros | version

version:    "vrsn:" /\d\d[.]\d\d/<reject: do{$item[2]>3.62} >
       |    <fatal: Cannot read files for versions after 3.62>

layout:     # etc.
```

- Now, as soon as an unsuitable version was encountered parsing would cease at once.

- There is also a <fatal?> directive that provides the equivalent to a fatal <error?>.

# Debugging grammars

- When grammars get large, or complex, or subtle, they may cease to work in the way we expect.

- RecDescent provides a range of tools that can assist in debugging a misbehaving grammar.

- These fall into two categories: tools to debug parser construction, and tools to debug actual parsing.

- All the tools are activated (or in one case, deactivated) by specifying a directive before the first rule of the grammar).

# Debugging parser construction

- The most obvious aid to debugging parser construction is the <warn> directive, which causes RecDescent to emit warnings about suspect grammatical constructions and likely sources of error, such as…

  - "Orphaned" rule items that appear after an <error> directive:

```
rule: option_1
    | <error> option_2
```

  - Missing rule definitions.

  - Instances where greedy repetition behaviour will almost certainly cause the failure of a production:

```
copy:  "cp"  filename(s)  filename
```

  - Attempts to define rules named Precompile, Save, Replace, or Extend.

  - Productions which consist of a single <error?> directive, and which therefore may succeed unexpectedly.

- Multiple consecutive lookahead specifiers:

```
rule:  …! … …! subrule  another_subrule
```

- Productions that start with a <reject> or <rulevar:...> directive and which have been optimized away.

- When rules are autogenerated by <autostub> (see page 33).

- The problems these warnings highlight are of varying degrees of severity (1, 2, or 3), so the <warn> directive can optionally take an integer argument that specifies the minimum level of severity that should be shown.

- RecDescent will provide extra diagnostic and explanatory information if the <hint> directive is also specified before the first rule.

- For example, this grammar:

```
<warn:2>          # Problems of medium
                  # or greater severity
rule: option_A
    | <error> option_B
```

produces the warning:

Warning: found a subrule (option_B) after an unconditional <error>.

- Whereas, with the <hint> directive:

```
<warn:2>
<hint>
rule: option_A
    | <error> option_B
```

the warning is:

Warning: found a subrule (option_B) after an unconditional <error>.

(Hint: An unconditional <error> always causes the production containing it to immediately fail. A subrule that follows an <error> will never be reached. Did you mean to use <error?> instead?)

- RecDescent also automatically performs several other compile-time checks (e.g. for left-recursion) that are important during development, but are an unnecessary burden in fully tested production code.

- If the `<nocheck>` directive is included in a grammar (before the first rule), these automated checks are skipped when a parser is built from the grammar.

- Sometimes, however, all of these diagnostics are insufficient, and you need to watch the parser being generated item-by-item.

- If the directive `<trace_build>` is specified before the first rule in the grammar, RecDescent emits debugging information that describes every rule, subrule, terminal, directive, and action that the parser generator finds in the grammar.

- In addition, if `<trace_build>` is specified, RecDescent dumps the entire parsing code it has generated to a file named `./RD_TRACE`.

# Debugging parser execution

- The `<trace_parse>` directive complements `<trace_build>`, providing detailed step-by-step tracking of the parsing process.

- If a `<trace_parse>` is specified before the first rule, the resulting parser dumps a trace of subsequent parses in a multicolumn column format.

- That format indicates the current rule being matched, the process of matching individual items, and the current context of the input string.

- The directive also takes an optional integer argument that specifies how many characters of context to display (by default, all the remaining input is shown).

# Parsing ambiguous input

- Sometimes the default rule matching mechanism (“first matching production wins”) isn't adequate.

- Consider the following data:

```
c,o,m,m,a,s,e,p,a,r,a,t,e,d
c:o:l:o:n:s:e:p:a:r:a:t:e:d
t   a   b   s   e   p   a   r   a   t   e   d
m,u:l,t:i,separ:a,ted
m:u:l,t:i,separ:a,ted
```

- Suppose we need to handle all three types of separators:

```
line: <skip:"">
    (   <leftop: value "," value>
     |  <leftop: datum ":" datum>
     |  <leftop: field "\t" field>
    )

value: /[^,]*/
datum: /[^:]*/
field: /[^\t]*/
```

- This rule works, but implicitly favours commas above colons above tabs.

- What if the correct separator is always the one that appears most frequently in the line? In other words:

```
(m),(u:l),(t:i),(separ:a),(ted)
(m):(u):(l,t):(i,separ):(a,ted)
```

- Recall that if a `<rulevar:...>` directive appears anywhere in any production of a rule, the named variable is declared (as a `my` variable, by default) at the start of the rule.

- In other words, the `<rulevar:...>` directive gives a rule “state”.

- Combined with rejection, that makes it easy to mess with the parser's mind, to allow it to tolerate ambiguity:

```
line: <rulevar: $max>
line: <rulevar: $maxcount=0>

line:  <leftop: value "," value>
           { $maxcount = @{$max = $item[1]}
                 if @{$item[1]} > $maxcount }
           <reject>
     |  <leftop: datum ":" datum>
           { $maxcount = @{$max = $item[1]}
                 if @{$item[1]} > $maxcount }
           <reject>
     |  <skip:""> <leftop: field "\t" field>
           { $maxcount = @{$max = $item[1]}
                 if @{$item[1]} > $maxcount }
           <reject>
     |  { $return = $max }

value: /[^,]*/
datum: /[^:]*/
field: /[^\t]*/
```

# Automating non-deterministic rules

- Although building a non-deterministic parser in this way is cute, it's also tedious and ugly.

- So RecDescent provides a mechanism to automate the process.

- The `<score:…>` directive takes a numerical argument that specifies how “well” the current production matches.

- The directive itself acts as an unconditional `<reject>`, so subsequent productions will also be tried.

- However, if no other production of the rule is accepted, the scored production with the highest numerical score will be deemed to have matched (i.e. as if the `<score:…>` hadn't been present).

- Hence, the multi-separator example could be re-written:

```
line:  <leftop: value "," value>
                    <score: @{$item[1]}>
    |  <leftop: datum ":" datum>
                    <score: @{$item[1]}>
    |  <skip:""> <leftop: field "\t" field>
                    <score: @{$item[2]}>

value: /[^,]*/
datum: /[^:]*/
field: /[^\t]*/
```

- This mechanism for exploring alternative interpretations is particularly useful when parsing natural language:

```
sentence: verb noun adj article noun
          { [@item] } <score: sensible(@item)>

        | adj noun verb article noun
          { [@item] } <score: sensible(@item)>

        | noun verb adj article noun
          { [@item] } <score: sensible(@item)>
```

- Given the sentence “Time flies like an arrow”, this rule would parse it all three ways and select the most sensible interpretation.

- It's also easy to reverse the sense of scoring, so that the production with the lowest score is always accepted:

```
line:  <leftop: value "," value>
                    <score: -@{$item[1]}>
    |  <leftop: datum ":" datum>
                    <score: -@{$item[1]}>
    |  <skip:""> <leftop: field "\t" field>
                    <score: -@{$item[2]}>

value: /[^,]*/
datum: /[^:]*/
field: /[^\t]*/
```

# Generalizing non-deterministic rules

- As the above examples indicate, rules that use scoring typically consist of a series of productions all of which end in exactly the same <score…> directive.

- That's tedious, so RecDescent allows us to specify a single scoring directive that is to be applied to every production in a rule:

```
line:  <autoscore: @{$item[-1]}>
    |  <leftop: value "," value>
    |  <leftop: datum ":" datum>
    |  <skip:""> <leftop: field "\t" field>
```

- The <autoscore:…> directive causes a <score:…> directive with the same argument to be automatically appended to every production (except any that already end in a <score:…>).

- The directive itself is the equivalent of a <reject>

# Automatic grammar generation

- Parse::RecDescent makes writing grammars much easier than *yacc*, but there are still aspects that can become repetitive, and hence tedious.

- For example, developing a grammar incrementally is tiresome, because you need to have the full grammar in place before you can test it.

- In some grammars, the same action is required in almost every production of every rule. Even cutting-and-pasting gets wearisome.

- And often you really just want the grammar to produce a full parse tree for the input. Repeatedly setting up the infrastructure to accomplish that is irksome.

- Fortunately, RecDescent provides automations to cover all these cases.

# Autostubbing

- Parse::RecDescent supports top-down design of grammars, with the `<autostub>` directive.

- Normally, if you create a grammar with one or more undefined rules:

```
my $grammar = q
{
  file:   defn(s)

  defn:   "def" ident ":" block

  block:  "{" item(s) "}"
};
```

  then RecDescent warns you (under the `-w` flag) when it compiles the parser.

- It then treats any missing subrule (such as `ident` or `item`) as an immediate, unconditional `<reject>`.

- This means that to develop and test a grammar top-down, you have to keep adding in dummy "stub" rules:

```
my $grammar = q
{
  file:   defn(s)

  defn:   "def" ident ":" block

  block:  "{" item(s) "}"

  ident:  "ident"

  item:   "item"
};
```

- This is not especially onerous at the higher levels – when there may be only one or two stubs – but later in the development process – when hundreds of stubs may be required – it's intolerable.

- That's where the `<autostub>` directive comes in.

- If this directive appears anywhere in a grammar, it cause Parse::RecDescent to automatically generate a `rule: "rule"` rule for any undefined rules it detects in a grammar.

# Autoactions

- There is a very common parsing task that always requires the same action at the end of a rule: parse-tree construction.

- There is also a large class of (normally operator-based) languages that are so regular in semantics that almost every rule in their grammar ends in the same action.

- Consider the task of translating infix notation to RPN. That is, given an expression like:

  ```
  1 && 2 + 3 * 4 + 5 ** 6 || 2 % 3 - 4 && 5
  ```

  produce:

  ```
  1 2 3 4 * + 5 6 ** + && 2 3 % 4 - 5 && ||
  ```

- The solution looks like this:

```
use Parse::RecDescent;

sub Parse::RecDescent::rpn
{
        for (my $i=1; $i<@list; $i+=2)
        {
                @_[$i,$i+1] = @_[$i+1,$i];
        }
        join " ", @_;
}

$grammar =
q{
        expr   :  <leftop: conj /(\|\|)/ conj>
                                { rpn(@{$item[1]}) }

        conj   :  <leftop: addn /(&&)/ addn>
                                { rpn(@{$item[1]}) }

        addn   :  <leftop: mult /([+-])/ mult >
                                { rpn(@{$item[1]}) }

        mult   :  <leftop: expo /([*\/\%])/ expo >
                                { rpn(@{$item[1]}) }

        expo   :  <leftop: unary /(\^|\*\*)/ unary >
                                { rpn(@{$item[1]}) }

        unary :   "(" expr ")"   { $item[2] }
              |   value          { $item[1] }

        value :   /\d+(\.\d+)?/  { $item[1] }
};

$parse = new Parse::RecDescent($grammar);

print $parse->expr($_), "\n" while <>;
```

- Note that the actions for the first four rules are identical.

- In fact, if there were more levels of precedence (e.g. & and | and and and or), those levels would have the same general structure and the same final action.

- RecDescent allows us to specify an "autoaction" – an action that is to be automatically added to the end of each production.

- Such an autoaction is appended to every production of every rule that does not already end in an explicit action.

- So we could rewrite the RPN translation grammar like this:

```
$grammar =
q{
        <autoaction: { rpn(@{$item[1]}) } >

        expr   :  <leftop: conj /(\|\|)/ conj>
        conj   :  <leftop: addn /(&&)/ addn>
        addn   :  <leftop: mult /([+-])/ mult >
        mult   :  <leftop: expo /([*\/\%])/ expo >
        expo   :  <leftop: unary /(\^|\*\*)/ unary >

        unary  :  "(" expr ")"   { $item[2] }
               |  value         { $item[1] }

        value  :  /\d+(\.\d+)?/  { $item[1] }
};
```

- Now the `expr`, `conj`, `addn`, `mult`, and `expo` rules have the action that was specified in the `<autoaction:...>` directive automatically appended to them.

- But because the productions of the `unary` and `value` rules already have trailing actions, those actions are used instead.

- Now let's consider the task of parsing a simple subset of LaTeX:

```
\documentclass[a4paper,11pt]{article}
\usepackage{latexsym}
\author{D. Conway}
\title{Parsing \LaTeX{}}
\begin{document}
\maketitle
\tableofcontents
\section{Description}
...is easy \footnote{But not \emph{necessarily} simple}.
\end{document}
```

- The grammar required is:

```
file:    directive(s)
            { bless \%item, "file" }

directive:  command
            { bless \%item, "directive" }
         |  literal
            { bless \%item, "directive" }

command: "\\" literal options(?) args(?)
            { bless \%item, "options"} }

options: "[" option(s? /,/) "]"
            { bless \%item, "options"} }

args:    "{" directive(s?) "}"
            { bless \%item, "args" }

option:  /[^][\\$&%#_{}~^ \t\n,]+/
            { bless \%item, "option" }

literal: /[^][\\$&%#_{}~^ \t\n]+/
            { bless \%item, "literal" }
```

- Notice that the action is really the same in every rule. Namely:

```
{ bless \%item, $item{__RULE__} }
```

- So we can use an autoaction here too:

```
<autoaction: { bless \%item, $item{__RULE__} } >

file:       directive(s)

directive:  command
          |  literal

command:    "\\" literal options(?) args(?)

options:    "[" option(s? /,/) "]"

args:       "{" directive(s?) "}"

option:     /[^][\\$&%#_{}~^ \t\n,]+/

literal:    /[^][\\$&%#_{}~^ \t\n]+/
```

# Autotrees

- Using a grammar to build a parse tree for an input is so common that RecDescent provides an even easier way to do it:

```
<autotree>

file:       directive(s)
directive:  command
          | literal
command:    "\\" literal options(?) args(?)
options:    "[" option(s? /,/) "]"
args:       "{" directive(s?) "}"
option:     /[^][\\$&%#_{}~^ \t\n,]+/
literal:    /[^][\\$&%#_{}~^ \t\n]+/
```

- The <autotree> directives acts just like the previous <autoaction:...> directive in that it appends an action to any production that doesn't aready end in one:

```
{ bless \%item, $item{__RULE__} }
```

- The only difference is that it appends a slightly different action to productions consisting of a single terminal (i.e. a pattern or literal string):

```
{ bless {__VALUE__ => $item[1]}, $item{__RULE__} }
```

- This makes it easier to handle terminals in a consistent manner.

- For example, we could write a simple object-oriented tool that summarizes the structure of a LaTeX file:

```
use Parse::RecDescent;

my $parser = Parse::RecDescent->new(<<'EOGRAMMAR');

     <autotree>

     file:        element(s)
     element:     command | literal
     command:     "\\" literal options(?) args(?)
     options:     "[" <leftop: option /,/ option>(s?) "]"
     args:        "{" element(s?) "}"
     option:      /[^][\\$&%#_{}~^ \t\n,]+/
     literal:     /[^][\\$&%#_{}~^ \t\n]+/

EOGRAMMAR

($parser->file(join "", <>)||die "Bad LaTeX")->explain(0)
```

```
sub file::explain {
        my ($self, $level) = @_;
        $_->explain($level) for @{$self->{element}}
}

sub element::explain {
        my ($self, $level) = @_;
        ($self->{command}||$self->{literal})->explain($level)
}

sub command::explain {
        my ($self, $level) = @_;
        print "\t"x$level,
              "Command: $self->{literal}{__VALUE__}\n";
        print "\t"x$level, "\tOptions:\n";
        $self->{options}[0]->explain($level+2)
                    if @{$self->{options}};
        print "\t"x$level, "\tArgs:\n";
        $self->{args}[0]->explain($level+2)
                    if @{$self->{args}};
}

sub options::explain
{
        my ($self, $level) = @_;
        $_->explain($level) foreach @{$self->{__DIRECTIVE1__}};
}

sub args::explain {
        my ($self, $level) = @_;
        $_->explain($level) foreach @{$self->{element}};
}

sub option::explain {
        my ($self, $level) = @_;
        print "\t"x$level, "Option: $self->{__VALUE__}\n";
}

sub literal::explain {
        my ($self, $level) = @_;
        print "\t"x$level, "Literal: $self->{__VALUE__}\n";
}
```

# Generic rules

- Another way in which RecDescent can automate grammars is by allowing us to specify generic rules.

- In a normal rule, the subrules that will be tried are fixed when the grammar is compiled.

- For example, a parser for a programming language might contain rules like this:

```
statement:  "while"  while_body  "end while"
          |  "if"     if_body     "end if"
          |  "func"   func_body   "end func"
          |  "proc"   proc_body   "end proc"
          |  "type"   type_body   "end type"
```

- The structural similarity of each of the productions is a clear indicator that something generic is happening.

- Factoring out the commonality of the "end..." strings is easy, since RecDescent allows us to use interpolated literals as terminals:

```
statement:  "while"  while_body   "end $item[1]"
          |  "if"     if_body     "end $item[1]"
          |  "func"   func_body   "end $item[1]"
          |  "proc"   proc_body   "end $item[1]"
          |  "type"   type_body   "end $item[1]"
```

- If there were then some way to select what type of ..._body subrule to call for each keyword, we could rewrite the entire rule:

```
statement:  keyword
            # something clever here
            "end $item[1]"

keyword:    "while" | "if" | "func" | "proc" | "type"
```

- The missing cleverness is provided by the <matchrule:...> directive.

- It treats its argument as an interpolated string that specifies which subrule to call:

```
statement:  keyword
            <matchrule: $item[1]_body >
            "end $item[1]"

keyword:    "while" | "if" | "func" | "proc" | "type"
```

- The interpolated subrule name is interpolated only when the directive is matched, so you can set up self-modifying selectors.

- For example: suppose we needed to identify genetic sequences of more than three consecutive bases where the transitions between bases followed the rule:

A ↔ C ↔ G ↔ T

- That is, given a sequence:

AAACTTTAAAACGTGCGCACGTGTAAAAAA

we want to locate:

AAACTTTAAAACGTGCGCACGTGTAAAAAA

- Here's a grammar that does that:

```
<autoaction: { $last = $item[1] } >

sequence:  <rulevar: local $last>

sequence:  base <matchrule:after_$last>(3..)
               { $return = [ $item[1], @{$item[-1]} ] }
         | base sequence
               { $return = $item[1] }

base:      /[ACGT]/

after_A:   /[C]/
after_C:   /[AG]/
after_G:   /[CT]/
after_T:   /[G]/
```

- The sequence matcher grabs the first base (automatically setting `$last`).

- It then attempts to match 3 or more repetitions of the subrule whose name is `"after_$last"`, with the subrule name being re-interpolated for each repetition.

- Each successful `after_`*`something`* match automatically resets `$last`, so that the next interpolation will select the appropriate successor.

- In a sense, the repeated `<matchrule:...>` directive is acting like a small finite state machine, embedded in the grammar.

- If the interpolated parsing fails to find three consecutive bases of the right types, the first production of the sequence rule will also fail.

- In that case, the second production will be invoked and will match a single base and then try the sequence again one step further down the input.

- This general pattern:

```
rule: subrule_A subrule_B
    | subrule_A rule
```

  is often useful to simulate a non-greedy prefix:

```
rule:  subrule_A(s?) subrule_A subrule_B
```

# Parametric rules

- The `<matchrule:…>` directive is a useful way of generalizing subrule *calls*, but by itself is not sufficient to provide fully generic rules.

- Consider, for example, the RPN translator shown on page 36:

```
expr  : <leftop: conj   /(\|\|)/     conj >
                                       { rpn(@{$item[1]})}
conj  : <leftop: addn   /(&&)/       addn >
                                       { rpn(@{$item[1]})}
addn  : <leftop: mult   /([+-])/     mult >
                                       { rpn(@{$item[1]})}
mult  : <leftop: expo   /([*\/\%])/  expo >
                                       { rpn(@{$item[1]})}
expo  : <leftop: unary  /(\^|\*\*)/  unary >
                                       { rpn(@{$item[1]})}

unary : "(" expr ")"                   { $item[2] }
      | /\d+(\.\d+)?/                  { $item[1] }
```

- The first five rules are structurally identical, consisting of a `<leftop:…>` directive followed by a common action.

- In other words, they are all really:

```
operation: <leftop:
              <matchrule: $operand >
              /($operator)/
              <matchrule: $operand >
           >
           { rpn(@{$item[1]}) }
```

- If there were some way to specify the values of `$operand` and `$operator` for each of the original rules, then the grammar could be greatly simplified.

- There is such a way: any subrule can be passed a set of arguments, which are then available within the subrule via the `@arg` array.

- So the operation rule could be implemented as:

```
operation: <leftop:
              <matchrule: $arg[0] >
              /($arg[1])/
              <matchrule: $arg[0] >
           >
           { rpn(@{$item[1]}) }
```

- Then the grammar would become:

```
expr  : operation[ "conj",  '\|\|'    ]
conj  : operation[ "addn",  '&&'      ]
addn  : operation[ "mult",  '[+-]'    ]
mult  : operation[ "expo",  '[*\/\%]' ]
expo  : operation[ "unary", '\^|\*\*' ]

unary : "(" expr ")"                   { $item[2] }
      | /\d+(\.\d+)?/                  { $item[1] }
```

- Arguments to subrules are passed in square brackets immediately after the subrule name.

- If the subrule is repeated, the argument brackets come before the repetition parentheses:

```
vnum: "v" listof[ "digits", "." ](1..3)
```

- In a complicated parametric rule like `operation`, positional arguments can be obscure, so RecDescent also allows named arguments to be passed via the `%arg` hash:

```
operation:
      <leftop:
        <matchrule: $arg{args} >
        /($arg{op})/
        <matchrule: $arg{args} >
      >                                  { rpn(@{$item[1]}) }

expr  : operation[ op => '\|\|',    args => "conj" ]
conj  : operation[ op => "&&",      args => "addn" ]
addn  : operation[ op => "[+-]",    args => "mult" ]
mult  : operation[ op => '[*\/\%]', args => "expo" ]
expo  : operation[ args => "unary", op => '\^|\*\*' ]

unary : "(" expr ")"                  { $item[2] }
      | /\d+(\.\d+)?/                 { $item[1] }
```

# Handling distributed text

- Many languages and mark-up systems have a facility for including code or text from other modules or files.

- Such facilities generally fall into one of two categories: *recursively processed* or *textually substituted*.

- Perl's `use ModuleName` and `do "filename"` functions are examples of recursively processed inclusion mechanisms.

- Using a module or doing a filename causes Perl to run its compiler on the corresponding file and add the *compiled* code to the program that issued the inclusion.

- In contrast, the C preprocessor `cpp` provides C and C++ with a textual substitution mechanism.

- Including a file via a `#include` causes the text of that file to be interpolated into the current program before the compiler even starts.

- Recursive evaluation is a more restrictive form of inclusion, but also safer.

- Suppose we wanted to parse a Lisp dialect that provided both mechanisms. (Anyone who is seriously interested in Lisp-via-Perl should see Gisle Aas' perl-lisp-0.05 bundle – on the CPAN.)

- This dialect might offer an `include` command that takes a filename argument and processes that file recursively.

- It might also offer an `interpolate` command that takes a filename argument and interpolates the text of that file into the current program *before* it is compiled.

- For example:

```
(defun max (a b) (if (< a b) b a))

(include 'sequence.lll )

(defun errormsg (quote (interpolate 'stdmsg_txt)))

(defun factorial (n)
  (if (< n 0)
                (errormsg)
                (* (series 2 n))))
```

- When the interpreter encounters the `include` in the second clause, it will locate the file "sequence.lll", compile its contents separately, and add the resulting expression tree to the program's expression tree.

- In contrast, when the interpreter encounters the `interpolate` in the third cons, it will locate the file "stdmsg_txt", replace the `interpolate` clause with the text in that file, and continue interpreting as if the file text had been there instead of the `interpolate` clause.

# The basic Lisp-like language parser

```
use Parse::RecDescent;

my $grammar = q
{
     program:  clause(s)

     clause:   left item(s) right      { $item[2] }
             |   <error>

     left:     "("

     right:    ")"

     item:     quote | atom | clause

     quote:    /'([^]()\s]+)/          { $1 }

     atom:     /[^]()\s]+/

};

my $parser = Parse::RecDescent->new($grammar);

use Data::Dumper;
print Data::Dumper->Dump($parser->program(join "", <>));
```

# Superbrackets

- Matching the parentheses at the end of a complex expression is tedious.

- To encourage laziness, many Lisp dialects provide a "superbracket" construct: a single square right bracket that takes the place of an arbitrary number of closing parentheses.

- So instead of:

```
(defun factorial (n)
  (if (< n 0)
                (errormsg)
                (* (series 2 n))))
```

  we can write:

```
(defun factorial (n)
  (if (< n 0)
                (errormsg)
                (* (series 2 n]
```

- Parsing such a construct presents a challenge, since a single square bracket now has satisfy the `right` subrule of an arbitrary number of nested calls to the rule:

  ```
  clause: left item(s) right
  ```

- The solution is to force the parser not to consume the square bracket whenever it matches a `right` subrule, and then consume it only when the outermost nested `clause` has been satisfied:

- ```
  my $grammar = q
  {
      program:      clause(s)

      clause:       cons super(?)           { $item[1] }
            |       <error>

      cons:         left item(s) right      { $item[2] }

      left:         "("

      right:        ")" | ...super

      super:        "]"

      item:         quote | atom | clause

      quote:        /'([^]()\s]+)/          { $1 }

      atom:         /[^]()\s]+/
  };
  ```
  ```

# Processing file inclusions recursively

- Now we're ready to add recursive file processing.
- The key is to note that Parse::RecDescent produces full re-entrant object-oriented parsers.
- Within the actions of a grammar, the parser itself can be accessed via the `$thisparser` object.
- So, if we have a rule that parses inclusions, there's no reason we can't call the parser recursively to process the text of the included file.
- The result of that recursive call then becomes the return value of the rule parsing the `include`.

```
my $grammar = q
{
      program:      clause(s)

      clause:       include
            |       cons super(?)          { $item[1] }
            |       <error>

      cons:         left item(s) right      { $item[2] }

      left:         "("

      right:        ")" | ...super

      super:        "]"

      item:         quote | atom | clause

      quote:        /'([^]()\s]+)/          { $1 }

      atom:         /[^]()\s]+/

      include:      left "include" quote right
                        { local (*FILE, $/);
                          open FILE, $item{quote}
                            or die "Can't include $item{quote}";
                          $return = $thisparser->program(<FILE>);
                        }
};
```

# Processing file inclusions by substitution

- The actual text that a parser is parsing is also directly available in actions, via the variable `$text`.

- If we change the contents of `$text` in a *successful* subrule, that change is preserved for the remainder of the parse.

- If the rule that changes `$text` is unsuccessful, then the change is lost.

- So, to provide textual substitution on the text being parsed, we simple arrange for the rule that parses interpolations to replace the interpolation with the contents of the specified file.

- We then continue to parse as if nothing had happened, and the interpolated text is processed.

```
my $grammar = q
{
   program:      clause(s)

   clause:       include
         |       interpolate item
         |       cons super (?)          { $item[1] }
         |       <error>

   cons:         left item(s) right       { $item[2] }

   left:         "("

   right:        ")" | ...super

   super:        "]"

   item:         quote | atom | clause

   quote:        /'([^]()\s]+)/           { $1 }

   atom:         /[^]()\s]+/

   include:      leftbracket "include" quote rightbracket
                     { local (*FILE, $/);
                       open FILE, $item{quote}
                         or die "Can't include $item{quote}";
                       $return = $thisparser->program(<FILE>);
                     }

   interpolate:  left "interpolate" quote right
                     { local (*FILE, $/);
                       open FILE, $item{quote}
                         or die "Can't interpolate $item{quote}";
                       $text = <FILE> . $text;
                     }
};
```

- Note that we need to explicitly parse an `item` once the file has been interpolated.

# Self-modifying grammars

- One of the most powerful features of Parse::RecDescent is the ability to specify grammars that change themselves on-the-fly.

- This is useful, for example, to help provide semantic feedback to a lexer (e.g. C++ declarations).

# Extending a rule

- Parsing C++ is fraught with peril.

- Once particularly nasty problem is that in some contexts there is no syntactic way to tell whether an identifier is a name or a typename.

- So a construct like this is inherently syntactically ambiguous:

```
eat barts (shorts);
```

- If `shorts` is a known typename, this is a function declaration; if `shorts` is a variable name it's a variable definition and initialization.

- Typically this is handled by fiat: “If it looks like an object initialization, it's treated as an object initialization, even if it's really a function declaration.”

- Alternatively, the parser can populate a table of known typenames, that the lexer then consults.

- In RecDescent that's relatively easy, since the lexer and parser are integrated:

```
start: { %{$thisparser->{types}} =
             ( int  => 1, float  => 2,
               void => 3, double => 4, ) }
       statement(s)

typename:
     identifier
       <reject:!$thisparser->{types}{$item[1]}}>

func_decl:
     typename identifier "(" typename ")"

var_init:
     typename identifier "(" identifier ")"

type_decl:
     "typedef" typename identifier
        { $thisparser->{types}{$item[3]} = $item[2] }
```

- But it's still ugly (and even uglier if you use a *yacc*-like parser!)

- Parse::RecDescent makes another approach possible: modifying the grammar itself whenever a type name is identified.

- Now, rather than replace a rule, we're going to extend one with additional productions:

```
typename:   # initially only 5 built-in typenames
    "int" | "char" | "float" | "double" | "void"

func_decl:
  typename identifier "(" typename ")"

var_init:
  typename identifier "(" identifier ")"

type_decl:
  "typedef" typename identifier
      { $thisparser->Extend("typename: '$item[3]'") }
```

# Grammars that learn

- In a sense, each time the previous grammar parses a `typedef` statement, it teaches itself to recognize the new type as a typename, by adding a new production to the `typename` rule.

- This kind of learning can easily be generalized to other tasks.

- The key is to anticipate the types of statements or questions that the grammar might be asked to parse.

- Then we generate rules that successfully parse those statements or questions, and whose resulting value is an acknowedgement or answer.

- Suppose, for example, we wanted to be able to teach a program about category hierarchies.

- We might use a self-modifying grammar, like this:

```
use Parse::RecDescent;

my $grammar = join("", <DATA>);

my $learner = Parse::RecDescent->new($grammar) || die;

$learner->interpret($_) while <>;

package Parse::RecDescent;
use Lingua::EN::Inflect ":ALL";

sub learn
{
  my ($self, %arg) = @_;
  my $new = $arg{unknown_thing};
  my $old = $arg{known_thing};

  my $new_grammar = qq {
          $old:                 $new
          $new:                 '$new'
          is_a_something:       'is' a '$new' { A('$old') }
          is_one_thing_another:'is' a '$new' a '$old'
          known_fact:           a '$new' 'is' a '$old'
  };

  $self->Extend($new_grammar);
}
```

```
__DATA__

interpret:
           fact | question | { print "\t> Huh?\n" }

fact:
           known_fact
              { print "\t> So I'm told\n" }
    |      a unknown_thing "is' a known_thing
              { $thisparser->learn(%item); print "\t> Okay\n"     }
    |      a unknown_thing "is" a unknown_thing
              { @item{"unknown_thing","known_thing"} = @item[2,5] }
              { $thisparser->learn(%item)                         }
              { print "\t> Okay, but what's a $item[5]?\n"        }

question:
           "what" is_a_something "?"
              { print "\t> ...$item[2]\n" }
     |     is_one_thing_another "?"
              { print "\t> Yes\n" }
     |  /is\b.*?[?]/
              { print "\t> I don't know\n" }

relationship:
           /\S+/

is_a_something:
           "is" "a" "thing" { "a thing" }

is_one_thing_another:
           "is" "a" "thing" "a" "thing"

known_fact:
           "a" "thing" "is" "a" "thing"

known_thing:
           thing

thing:
           "thing"

unknown_thing:
           /\S+/

a:
           /an?/
```

- This program has two serious problems.

- Firstly, it only understands a single type of relationship ("is a"), so it can't be taught other types of facts.

- Secondly, it has no persistence mechanism, so when the program finishes, it forgets everything it was taught.

- To overcome the first problem, we have to generalize the way the grammar represents relationships.

- To overcome the second problem, we have to save the grammar when the program exits, and restore it next time the program starts.

- The following version solves both these problems, and allows the grammar to generalize its understanding from a single case:

```
my $grammar = join("", <DATA>);

my $learner =  (eval { require "Knowledge.pm" })
        ? Knowledge->new()
        : Parse::RecDescent->new($grammar) || die;

$learner->interpret($_) while <>;

$learner->Save("Knowledge");

package Parse::RecDescent;
use Lingua::EN::Inflect ":ALL";

sub learn {
    my ($self, $sub, $verb, $obj) = @_;
    my ($sub_pl, $verb_pl, $obj_pl) =
        ( PL_N($sub), PL_V($verb), PL_N($obj) );

    my ($relates, $does_relate, $is) = ("")x3;

    if ($verb eq "is") {
        $is         = qq($obj: $sub\n $obj_pl: $sub_pl);
        $relates    = qq('is' a '$sub'   { A('$obj') }
                        |'are' '$sub_pl' { '$obj_pl' } );
        $does_rel   = qq('is' a $obj a '$obj'
                        | 'are' $obj_pl '$obj_pl'    );
    }
    else {
        $relates    = qq('$verb' a '$obj'     { A('$sub') }
                        |'$verb_pl' '$obj_pl' { '$sub_pl' }  );
        $does_rel   = qq('does' a '$sub' '$verb_pl' a '$obj'
                        |'do' '$sub_pl' '$verb_pl' '$obj_pl' );
    }

    my $grammar = qq{
        $sub: '$sub'
        $sub_pl: '$sub_pl'
        $is
        relates_to_something: $relates
        does_something_relate_to_something: $does_rel
        known_fact: a '$sub' '$verb' a '$obj'
                  | '$sub_pl' '$verb_pl' '$obj_pl'
    };

    $self->Extend($grammar);
}
```

```
__DATA__

interpret:
      fact | question | { print "\t> Huh?\n" }

fact:
      known_fact
         { print "\t> So I'm told\n" }

    | a known_thing relationship a known_thing
         { $thisparser->learn(@item[2,3,5]); print "\t> Okay\n" }

    | a unknown_thing relationship a known_thing
         { $thisparser->learn(@item[2,3,5]); print "\t> Okay\n" }

    | a known_thing relationship a unknown_thing
         { $thisparser->learn(@item[2,3,5]); print "\t> Okay\n" }

    | a unknown_thing relationship a unknown_thing
         { $thisparser->learn(@item[2,3,5])       }
         { print "\t> Okay, what's a $item[5]?\n" }

question:
      "what" relates_to_something "?"
         { print "\t> ...$item[2]\n" }

    | does_something_relate_to_something "?"
         { print "\t> Yes\n" }

    | /(does|is)\b.*?[?]/
         { print "\t> I don't know\n" }
```

```
relationship:
    /\S+/

relates_to_something:
    "is" "a" "thing" { "thing" }

does_something_relate_to_something:
    "is" "a" "thing" "a" "thing"

known_fact:
    "a" "thing" "is" "a" "thing"

known_thing:
    thing

thing:
    "thing"

unknown_thing:
    /\S+/

a:
    /an?/
```

# Persistence of memory

- In order to "remember" what it has learnt, the program uses the `Save` method to create a module (Knowledge.pm) that can be used to recreate the `$learner` object:

```
$learner->interpret($_) while <>;

$learner->Save("Knowledge");
```

- So now, when it runs, the program first has to check whether that module exists:

```
my $learner = (eval { require "Knowledge.pm" })
            ? Knowledge->new()
            : Parse::RecDescent->new($grammar) || die;
```

# (Nearly) parsing Perl

- "Only perl can parse Perl"

- But with some help from Text::Balanced, RecDescent can get arbitrarily close.

- RecDescent provides three directives that make the Perl parsing routines of Text::Balanced directly available in a grammar…

- `<perl_codeblock>`: this directive does an `extract_codeblock($text,"{}",$skip)` on the text being parsed. It returns the substring containing the extracted code block on success.

- `<perl_variable>`: this directive does an `extract_variable($text, $skip)` on the text being parsed. It returns the substring containing the extracted code block on success.

- `<perl_quotelike>`: this directive does an `extract_quotelike($text, $skip)` on the text being parsed. It returns a reference to an array containing the various components of the quote-like operation.

- For example, here's CSV with a twist of nacre:

```
my $PCSV_parser = Parse::RecDescent->new(<<'EOGRAMMAR');

  file:   <skip:'[ \t]*'> line(s?)

  line:   field(s? /,|=>/) "\n" { $item[1] }

  field:  <perl_quotelike>      { join "", @{$item[1]} }
       |  /((?!,|=>).)*/

EOGRAMMAR
```

- This parser accepts CSV lines where the fields may be quoted using "…", '…', q{…}, qq/…/, etc., and the separators are either type of Perl comma operator.

# Source code filtering

- The ability to parse a large subset of Perl comes in handy when you need to do source code filtering, since it's (presumably) some variant of Perl you're processing.

- Source code filtering isn't a particularly common task but, once you know how, it's astonishingly useful for sidestepping Perl's occasional limitations.

- Paul Marquess's Filter::Util::Call module makes it easy to grab (and mess with) the source of a Perl program before it reaches the compiler.

- In its simplest configuration, you first install the filter in the `import` subroutine of a module.

- Then you provide a filtering subroutine (named `filter`) that reads the program source into `$_`, and processes it.

- The Filter::Util::Call module then passes the final value in $_ to the compiler (or to any other filtering modules that are also in effect).

- Here's a simple filter module that takes every occurrence of "or" and replaces it with "||"

```
package ors2oars;
use Filter::Util::Call;

sub import   { filter_add({}) }

sub filter {
        my $status = filter_read();
        s/or/||/g unless $status<0;
        return $status;
}
```

- To use it, we'd write:

```
use ors2oars;

sub e { 2.71828 }

print 0 or e;

print sort(2,17,5,3)||e;

print "Wordsworth's horror oratory bored forlorn
       authors into torpor\n";
```

- Which would pre-translate the program to:

```
sub e { 2.71828 }

print 0 || e;

print s||t(2,17,5,3)||e;

print "W||dsw||th's h||r|| ||at||y b||ed f||l||n
       auth||s into t||p||\n";
```

- This illustrates the main problem with code filtering: to do it correctly, you need to *parse* the code you're modifying, so you know which bits to ignore.

- Of course, the example is silly, but the same problems arise in real filters too.

# When prototypes attack: the `winnow` function

- To see how useful source code filtering can be, let's consider a significant limitation of Perl: it's not possible to prototype `grep`-like subroutines.

- The builtin `grep` function takes as its first argument either a block, or an expression (typically a pattern or a variable).

- Suppose we wanted to write a `grep`-like subroutine called `winnow`, which returns two lists: one containing elements for which the block or expression was true; the other containing the remaining elements:

```
my ($good, $bad)        = winnow { good($_) } @ugly;
my ($hashed, $unhashed) = winnow { me=>1, you=>0}, @us;
my ($vowel, $novowel)   = winnow /[aeiou]/, @words;

print "The good were: @$good\n";
print "The bad were:  @$bad\n";
```

- There is currently no way of specifying a prototype for `winnow` that will handle all three types of first argument (code block, pattern, hash reference).

- But we can grab the source before the compiler sees it and avoid the problem entirely.

- To do that, we need to walk through the program's code and identify places where `winnow` is called.

- Then we check the first argument and insert enough extra syntax to convert it to a reference (i.e. add `"sub"` before blocks and `"qr"` before patterns).

- Then `winnow` becomes:

```
sub winnow
{
  my ($ref, @list) = @_;
  my $reftype = ref $ref;
  my $selector = ($reftype eq "Regexp") ? sub { /$ref/ }
               : ($reftype eq "HASH")   ? sub { $ref->{$_} }
               : ($reftype eq "CODE")   ? $ref
               : croak "Bad selector ($reftype) for &winnow";
  my (@sheep, @goats);
  foreach (@list)
  {
          if ($selector->()) { push @sheep, $_ }
          else               { push @goats, $_ }
  }
  return [\@sheep, \@goats];
}
```

- So how do we pre-munge the source?

# Winnowing Perl with Text::Balanced on Occam's Razor

- To pre-filter calls to `winnow` we need a filtering module that can distinguish such calls from other occurrences of the string "`winnow`" in the source:

```
use winnow;

my @list = qw( ant fly spider winnow );
push @list, q(winnow { /[aeiou]/ } @list);
$hash{winnow} = 1;
*win = *winnow;

use Data::Dumper;

print Data::Dumper->Dump(winnow { /[aeiou]/ } @list);
print Data::Dumper->Dump(winnow /[aeiou]/, @list);
print Data::Dumper->Dump(winnow { ant=>1, %hash }, @list);
```

- The easiest way to achieve that is step through the source, skipping over quotes, quote-likes, and variables, and treat any other occurrences of "`winnow`" as calls:

```
package winnow;

sub winnow {
     my ($ref, @list) = @_;
     my $reftype = ref $ref;
     my $selector = ($reftype eq "Regexp") ? sub { /$ref/ }
                  : ($reftype eq "HASH")   ? sub { $ref->{$_} }
                  : ($reftype eq "CODE")   ? $ref
                  : croak "Bad selector ($reftype) for &winnow";

     my (@sheep, @goats);
     foreach (@list)
     {
              if ($selector->()) { push @sheep, $_ }
              else               { push @goats, $_ }
     }

     return [\@sheep, \@goats];
}


use Filter::Util::Call;

sub import   { filter_add({}) }
sub unimport { filter_del() }


sub filter
{
     my $status = filter_read(10_000);

     return $status if $status<0;

     my $line = (caller)[2]+1;
     $_ = "#line $line\n" . rewinnow($_);

     return $status;
}
```

```
use Text::Balanced ":ALL";

sub rewinnow
{
    my $source = shift;
    pos $source = 0;
    my $text = "";
    my $nextbit;

    while (pos $source < length $source)
    {
        if ($source =~ /\G(\s+)/gc)
        {
            $text .= $1;
        }
        elsif (($nextbit) = extract_quotelike($source) and $nextbit
           or  ($nextbit) = extract_variable($source) and $nextbit)
        {
            $text .= $nextbit;
        }
        elsif ($source =~ m/\Gwinnow/gc)
        {
            $text .= "winnow::winnow";
            if ($source =~ m/\G(\s*[(]?\s*)(?=\/)/gc ||
                $source =~ m/\G(\s*[(]?\s*)m\b/gc )
            {
                $text .= $1 . "qr";
            }
            elsif (($nextbit) = extract_codeblock( $source,"{")
                   and $nextbit and $source=~m/\G(,?)/gc)
            {
                my $is_block = !$1;
                $text .= " sub " if $is_block;
                $text .= " " . rewinnow($nextbit). ",";
            }
        }
        else
        {
            $source =~ m/\G(\w+|#.*\n|\W)/gc;
            $text .= $1;
        }
    }
    return $text;
}
```

# Winnowing Perl with Parse::RecDescent

- Of course, there's no need to have such ugly parsing code.

- We could get the same effect much more cleanly using RecDescent:

```
sub filter
{
        my $status = 1;
        $status = filter_read(10_000);
        return $status if $status<0;
        my $line = (caller)[2]+1;
        $_ = "#line $line\n" . Parse::RecDescent::rewinnow($_);
        return $status;
}

use Parse::RecDescent;
package Parse::RecDescent;

my $transform = Parse::RecDescent->new(<<'EOGRAMMAR');

    program: <skip:""> chunk(s?)     { join "", @{$item[-1]} }

    chunk:   /\s+/
         |   <perl_quotelike>        { join "", @{$item[1]} }
         |   <perl_variable>
         |   "winnow" <skip:'\s*'> winnow_arg
                                     { "winnow::winnow $item[3]" }
         |   junk

    winnow_arg:
             /(\s*[(]?\s*)(?=\/)/    { "$1 qr" }
         |   /(\s*[(]?\s*)m\b/       { "$1 qr" }
         |   <perl_codeblock> /,?/   { ($item[2] ? " " : " sub ") .
                                       rewinnow($item[1]) . ","}
         |   ""

    junk:    /\w+|#.*\n|\W/

EOGRAMMAR

sub rewinnow { $transform->program($_[0]); }
```

# Metagrammars

- As far as a parser generator like RecDescent is concerned, a grammar is just another type of input text.

- There's nothing to prevent us from writing grammars to specify parsers for other grammars.

- Randal Schwartz put Parse::RecDescent to that very use in one of his Linux Magazine columns:
  `http://www.stonehenge.com/merlyn/LinuxMag/col04.html`

- Apparently tiring of writing clear, informative prose, he decided to build *spew* – a generator of meanful nonsense.

- *Spew* takes an input text that is itself a grammar.

- It then run this grammar "backwards", using it as a template to randomly generate sentences.

- For example, given the following grammar:

```
START:
  mission "\n\n" mission "\n\n" mission "\n"

mission:
  Our_job_is_to " " do_goals "." |
  2 @ Our_job_is_to " " do_goals " " because "."

Our_job_is_to:
  ("It is our " | "It's our ") job " to" |
  "Our " job (" is to" | " is to continue to") |
  "The customer can count on us to" |
  ("We continually "|"We ") ("strive"|"envision"|"exist") " to" |
  "We have committed to" | "We"

job:
  "business" | "challenge" | "goal" | "job" | "mission" |
"responsibility"

do_goals:
  goal | goal " " in_order_to " " goal

in_order_to:
  "as well as to" | "in order that we may" |
  "in order to" | "so that we may endeavor to" |
  "so that we may" | "such that we may continue to" |
  "to allow us to" | "while continuing to" | "and"

because:
  "because that is what the customer expects" |
  "for 100% customer satisfaction" |
  "in order to solve business problems" |
  "to exceed customer expectations" |
  "to meet our customer's needs" |
  "to set us apart from the competition" |
  "to stay competitive in tomorrow's world" |
  "while promoting personal employee growth"

goal: adverbly " " verb " " adjective " " noun
```

```
adverbly:
  "quickly" | "proactively" | "efficiently" | "assertively" |
  "interactively" | "professionally" | "authoritatively" |
  "conveniently" | "completely" | "continually" | "globally" |
  "enthusiastically" | "collaboratively" | "synergistically" |
  "seamlessly" | "competently" | "dramatically"


verb:
  "maintain" | "supply" | "provide access to" | "disseminate" |
  "network" | "create" | "engineer" | "integrate" | "restore" |
  "leverage other's" | "revolutionize" | "customize" | "build" |
  "negotiate" | "leverage existing" |  "facilitate" | "foster" |
  "initiate" | "enhance" | "simplify" | "pursue" | "utilize" |
  "administrate" | "promote" | "coordinate" | "fashion"


adjective:
  "inexpensive" | "timely" | "effective" | "unique" | "quality" |
  "cost-effective" | "virtual" | "scalable" | "error-free" |
  "professional" | "value-added" | "business" | "diverse" |
  "high-quality" | "competitive" | "excellent" | "innovative" |
  "corporate" | "high standards in" | "world-class" |
  "performance-based" | "multimedia-based" | "market-driven" |
  "cutting edge" | "high-payoff" | "low-risk high-yield" |
  "long-term high-impact" | "prospective" | "progressive" |
  "ethical" |"economically sound" | "parallel" | "emerging" |
  "enterprise-wide" | "principle-centered" | "mission-critical" |
  "resource-leveling" | "interdependent" |
  "seven-habits-conforming" |

noun:
  "content" | "paradigms" | "data" | "opportunities" |
  "information" | "services" | "materials" | "technology" |
  "benefits" | "solutions" | "infrastructures" | "products" |
  "deliverables" | "catalysts for change" | "resources" |
  "methods of empowerment" | "sources" | "leadership skills" |
  "meta-services" | "intellectual capital"
```

then *spew* might generate the following:

> We exist to conveniently administrate value-added services while continuing to globally customize virtual data in order to solve business problems.
>
> It's our responsibility to efficiently revolutionize business paradigms in order to enthusiastically integrate high-quality solutions.
>
> We have committed to authoritatively disseminate excellent solutions in order that we may proactively build multimedia-based products in order to solve business problems.
>
> Our challenge is to continue to seamlessly foster business materials in order that we may seamlessly negotiate effective paradigms while promoting personal employee growth.

- If we then gave *spew* the following grammar:

```
START:
     stanza stanza "\n\n" exclaim "\n\n" stanza exclaim

stanza:
     sentence comparison question
   | sentence comparison
   | comparison comparison exclaim
   | address question question sentence

sentence:
     "The " adjectiveNotHep " " personNotHep " " verbRelating
            " the " adjectiveHep " " personHep ". "
   | "The " personHep " " verbRelating " the " adjectiveNotHep
            ", " adjectiveNotHep " " personNotHep ". "

question:
     ques_start adjectiveHep " " personNotHep "? "
   | ques_start adjectiveNotHep " " personHep "? "

comparison:
     "One says '" compNotHep "'
           while the other says '" compHep "'. "
   | "One thinks '" compNotHep "'
           while the other thinks '" compHep "'. "
   | "They shout '" compNotHep "!' And we shout '" compHep "'. "
   | "It's " compNotHep " versus " compHep "! "

personNotHep:
     "capitalist" | "silk purse man" | "square" | "banker"
   | "Merchant King" | "pinstripe suit"

personHep:
     "cat" | "beat soul" | "wordsmith" | "hep cat" | "free man"
   | "street poet" | "skin beater" | "reed man"

adjectiveNotHep:
     "soul-sucking" | "commercial" | "cash-counting" | "uncool"
   | "bloody-handed" | "four-cornered" | "love-snuffing"
```

```
adjectiveHep:
      "love-drunk" | "cool, cool" | "happening" | "tuned-in"
    | "street wise" | "wise and learned"

verbRelating:
      "evades" | "fears" | "distresses" | "dodges" | "battles"
    | "curses" | "dislikes" | "begrudges" | "belittles" | "avoids"

compNotHep:
      "recreation" | "isolation" | "tranportation"
    | "complication" | "subordination" | "sacred nation"

compHep:
      "fornication" | "instigation" | "interpretation"
    | "elevation" | "animation" | "inebriation" | "true relation"

ques_start:
      "Could there ever be a "
    | "How could there be a "
    | "Can you picture a "

address:
      "Catch this: " | "Listen, cats, " | "Dig it: "
    | "I lay this on you: "

exclaim:
      "Heavy, man. "|"Heavy. " | "Yow! "
    | "Snap 'em for me. " | "Dig it. "
```

then it would turn its hand to Beat Poetry:

*One says 'subordination' while the other says 'fornication'. One says 'tranportation' while the other says 'inebriation'. Snap 'em for me. Listen, cats, Can you picture a love-drunk banker? How could there be a cash-counting skin beater? The commercial square dislikes the tuned-in wordsmith.*

*I lay this on you: How could there be a four-cornered cat? Could there ever be a street wise capitalist? The hep cat dodges the four-cornered, commercial banker. Heavy, man.*

*It's subordination versus inebriation! They shout 'sacred nation!' And we shout 'true relation'. Heavy, man. The free man dislikes the bloody-handed, cash-counting pinstripe suit. One says 'subordination' while the other says 'true relation'. How could there be a love-snuffing free man?*

*Yow!*

# Building a grammar for parsing grammars

- You can read the gory implementation details of *spew* in Randal's column, but the heart of it is a RecDescent grammar for a parser that parses *spew* grammars:

```
# Spew metagrammar.  By Randal Schwartz, 1999.

{ my %grammar; my $internal = 0; }

grammar: rule(s) /\Z/
           { \%grammar; }

rule:    identifier ":" defn
           { push @{$grammar{$item[1]}{is}}, @{$item[3]};
             $grammar{$item[1]}{defined}{$itempos[1]{line}{to}}++;
             $item[1]; }
    |    <error>

defn:    <leftop: choice "|" choice>

choice:  weight item(s)
           { [ $item[1] => @{$item[2]} ] }

weight:  /\d+(\.\d+)?/ /\@/
           { $item[1] } | { 1 }

item:    <perl_quotelike>
           { " " . $item[1][2] }
    |    identifier ...!/:/
           { $grammar{$item[1]}{used}{$itempos[1]{line}{to}}++;
             $item[1]; }
    |    "(" defn ")"
           { push @{$grammar{++$internal}{is}}, @{$item[2]};
             $internal; }
    | <error>

identifier: /[A-Za-z_]\w*/
```

# MAKE PUBLICATIONS FAST!!!

- In 1996, physicist Alan Sokal published a paper in the journal *Social Text*.

- It was entitled "Transgressing the Boundaries: Toward a Transformative Hermeneutics of Quantum Gravity".

- In it, he used a combination of "cultural studies" jargon and sheer scientific nonsense to argue that gravity is not a physical phenomenon, but rather a social construct to which we are all culturally conditioned.

- The hoax and the furore it subsequently created make for fascinating reading:
  `http://www.physics.nyu.edu/faculty/sokal/index.html`

- But what most commentators missed was the fact that with linguistic generators like *spew* and the gullibility of reviewers in the soft sciences, there is a remarkable opportunity to quickly and painless build oneself an impressive academic record.

- Andrew Bulhak, a postgraduate student at Monash University has built a grammar-based text generator – the Dada Engine – that extends the ideas suggested by *spew* several orders of magnitude.

- His test domain was the same as Sokal's: postmodernist social theory.

- So now you can download as many papers as you need to start your academic career from:
  `http://www.elsewhere.org/cgi-bin/postmodern/`

- Here's an example to launch your brilliant career:

# Narratives of Paradigm: Realism and the cultural paradigm of discourse

Jean-Luc O. Scuglia
Department of Sociology, Cambridge University

## 1. The cultural paradigm of discourse and textual precultural theory

"Society is part of the defining characteristic of truth," says Marx. The subject is contextualised into a realism that includes consciousness as a whole.

"Sexual identity is elitist," says Foucault; however, according to Buxton[1] , it is not so much sexual identity that is elitist, but rather the failure of sexual identity. It could be said that Lyotard uses the term 'the cultural paradigm of discourse' to denote the role of the reader as participant. The subject is interpolated into a Lacanist obscurity that includes narrativity as a paradox.

However, the main theme of la Fournier's[2] model of the cultural paradigm of discourse is the common ground between society and sexual identity. If realism holds, we have to choose between the cultural paradigm of discourse and neodeconstructive structuralist theory.

Thus, Bataille suggests the use of realism to challenge hierarchy. The cultural paradigm of discourse states that reality must come from the collective unconscious. However, a number of theories concerning postcultural materialism exist. Abian[3] holds that the works of Spelling are an example of self-supporting objectivism.

In a sense, the characteristic theme of the works of Spelling is a mythopoetical totality. In *Melrose Place*, Spelling deconstructs realism; in *The Heights* he examines the cultural paradigm of discourse.

## 2. Narratives of absurdity

The main theme of Finnis's[4] essay on realism is not discourse, as the cultural paradigm of discourse suggests, but postdiscourse. It could be said that Sontag's critique of textual precultural theory states that the media is fundamentally impossible, given that language is interchangeable with culture. The primary theme of the works of Spelling is the difference between society and sexual identity.

In the works of Spelling, a predominant concept is the distinction between closing and opening. But the subject is contextualised into a dialectic theory that includes language as a reality. If the cultural paradigm of discourse holds, we have to choose between textual precultural theory and subcapitalist rationalism.

If one examines dialectic precultural theory, one is faced with a choice: either accept the cultural paradigm of discourse or conclude that art is capable of significant form. In a sense, any number of semioticisms concerning the futility, and eventually the economy, of

constructive narrativity may be revealed. D'Erlette[5] implies that we have to choose between realism and subtextual socialism.

The main theme of Dietrich's[6] analysis of textual precultural theory is a cultural totality. Thus, the premise of predialectic desituationism states that truth serves to reinforce the status quo. Foucault promotes the use of the cultural paradigm of discourse to deconstruct class.

In a sense, Sontag's essay on textual precultural theory implies that academe is capable of significance. Marx uses the term 'realism' to denote the bridge between sexuality and class.

However, the characteristic theme of the works of Tarantino is a mythopoetical whole. Sontag suggests the use of textual precultural theory to attack sexism. In a sense, many materialisms concerning realism exist. If textual precultural theory holds, we have to choose between the cultural paradigm of discourse and the deconstructive paradigm of concensus.

Thus, the example of textual precultural theory which is a central theme of Tarantino's *Pulp Fiction* emerges again in *Four Rooms*. An abundance of depatriarchialisms concerning the defining characteristic, and subsequent meaninglessness, of postcultural society may be discovered.

It could be said that Foucault uses the term 'the cultural paradigm of discourse' to denote not narrative, but neonarrative. Von Ludwig[7] states that the works of Tarantino are not postmodern.

But Baudrillard promotes the use of textual precultural theory to read and deconstruct reality. Realism holds that art is used to oppress the proletariat, given that the premise of the cultural paradigm of discourse is invalid.

## References


1. Buxton, M. V. R. ed. (1975) *Dialectic situationism, realism and feminism*. Harvard University Press
2. la Fournier, M. B. (1992) *Reinventing Socialist realism: Realism in the works of Madonna.* O'Reilly & Associates
3. Abian, J. W. K. ed. (1974) *The cultural paradigm of discourse and realism*. University of Massachusetts Press
4. Finnis, H. F. (1995) *Reading Marx: Subcultural socialism, feminism and realism*. And / Or Press
5. d'Erlette, L. T. Q. ed. (1970) *The cultural paradigm of discourse in the works of Tarantino.* Loompanics
6. Dietrich, F. B. (1998) *The Burning House: Realism and the cultural paradigm of discourse.* Schlangekraft
7. von Ludwig, R. ed. (1987) *Realism in the works of Cage.* O'Reilly & Associates

# Parse::FastDescent (RecDescent 2.0)

- The Parse::RecDescent module is still under active development and the author is very responsive to user feedback.
- Speed improvements
- Cleaner interface (more fully object-oriented)
- Non-insatiable repetitions (i.e. backtracking)
- Parsimonious repetitions
- Text reconstruction ("anti-parsing") of autotreed grammars.
- User configurable error reporting mechanism
- Modular grammars (internal namespaces)