

use Parse::RecDescent;

una breve introduzione per esempi
sull'uso di Parse::RecDescent

*(ovvero come scrivere un parser SQL
in 40 minuti senza sapere nulla
di P::RD e senza bruciare la cucina)*

IPW2009 – Pisa

Francesco 'Oha' Rivetti <oha[at]oha.it>

Intro

Parse::RecDescent è un modulo Perl scritto da Damian Conway.

DATI SEQUENZIALI

+

GRAMMATICA

=

DATI COMPLESSI

... ma in pratica?

A cosa serve?

- Distinguere date fra vari formati
- Interpretare comandi per un bot IRC
- Modificare una DDL SQL seguendo uno schema di modifica
- Compilare un micro-linguaggio
- Analizzare codice sorgente
- Trasformare un wiki-text in HTML
- Tanti altri...

A cosa serve? - 2

Più in dettaglio con P::RD possiamo:

- **Separare** la logica di parsing dal programma
- **Uniformare** dati diversi
- Analizzare i contenuti in base al **contesto**

Ma prima vediamo cos'è una grammatica...

Grammatica

Una grammatica è un sequenza di **regole** che stabiliscono quali dati possono esser accettati

Ogni regola è definibile come una serie di possibili **produzioni**, ognuna delle quali può esser composta da regole, terminali, azioni o direttive.

Per $P::RD$ un **terminale** è una stringa o una espressione regolare, o più precisamente la foglia del nostro albero di parse.

Vediamo un esempio...

Grammatica - 2

```
rule : foo bar cuz  
      | left right
```

La regola `rule` ha 2 possibili produzioni che vengono tentate nell'ordine.

Ogni produzione fallisce a meno che tutte le sue regole restituiscano, nell'ordine, un risultato positivo.

Qualora tutte le regole restituiscano un valore positivo allora l'intera regola ritorna **l'ultimo valore** restituito.

Proviamo un esempio con dei dati di ingresso...

Esempio con dati

```
A : B
   | C
   | <error>
B : 'foo' C
C : 'bar'
```

L'esempio sopra definisce una grammatica – non particolarmente utile – composta da una regola **A** che può “accettare” le regole **B**, **C** oppure **<error>**

La regola **C** accetta **'bar'** mentre la regola **B** accetta **'foo'** seguita da **C**.

Esempio con dati - 2

#IN> bar

A : B | C | <error>

B : 'foo' C

C : 'bar'

Se in ingresso avessimo 'bar' per la regola **A** si tenterebbero le sue 2 produzioni:

- 1) **B fallisce** – richiede che la stringa contenga 'foo'!
- 2) **C** restituisce 'bar' ...
- 3) Quindi **A** restituisce a sua volta 'bar'

Esempio con dati - 3

#IN> foobar

A : B | C | <error>

B : 'foo' C

C : 'bar'

Se invece passassimo la stringa 'foobar':

- 1) B consuma il terminale 'foo' e prosegue tentando c
- 2) c consuma 'bar' e lo restituisce a B, che a sua volta lo restituisce ad A
- 3) A restituisce 'bar'

Esempio con dati - 4

#IN> other

A : B | C | <error>

B : 'foo' C

C : 'bar'

Se invece passassimo 'other' allora il nostro parser **fallirebbe le prime due produzioni** di A e ripiegherebbe su <error>, che ha l'inaspettato effetto di produrre un errore!

ERROR (line 1): Invalid A: Was expecting B, or C

(Gradevole l'errore leggibile, vero!?!)

Esempio Data

```
# "Sabato 19 Settembre 2009"
```

```
date : wday mday month year
```

```
wday : 'Lunedì' | 'Martedì' | 'Mercoledì' ...
```

```
mday : /(\d{1,2})/
```

```
month : 'Gennaio' | 'Febbraio' ...
```

```
year : /(\d{4})/
```

Esempio Data 2

```
# "Sabato 19 Settembre 2009" | "2009-09-24"  
date : wday mday month year  
      | year /-/ nmonth /-/ mday  
wday : 'Lunedì' | 'Martedì' | 'Mercoledì' ...  
mday : /(\d{1,2})/  
month : 'Gennaio' | 'Febbraio' ...  
nmonth : /\d\d?/  
year : /(\d{4})/
```

In rosso le aggiunte per fargli accettare anche un secondo “formato” di data.

Ma non si parlava di un modulo Perl?

Ecco come usare queste grammatiche in Perl:

```
my $grammar = <<'EOG' ;
```

```
A : B | C | <error>
```

```
B : 'foo' C
```

```
C : 'bar'
```

```
EOG
```

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

```
$parser->A('foobar') ;
```

Azioni

A : **B** | **C** | **<error>**

B : **'foo'** **C**

C : **'bar'**

Ogni volta che `Parse::RecDescent` risolve una regola, **produce l'ultimo valore della produzione**, che – per un terminale – equivale al testo consumato.

Nel caso già visto usando `'foobar'` come ingresso, abbiamo la regola `B` che consuma `'foo'` e tenta `C` che restituisce `'bar'`.

`B` restituisce ciò che consuma `C`, cioè `'bar'` !!!

Azioni - 2

```
A : B | C | <error>
```

```
B : 'foo' C { $item[1]. '-->' . $item[2] }; }
```

```
C : 'bar'
```

B ora restituirà ad A 'foo-->bar' invece di solo 'bar'.

@**item** contiene infatti i valori di ritorno di ogni subrule, nonché il nome della regola attuale in **\$item[0]**

Nota: un'azione è parte della produzione; **se restituisce un valore falso, annulla la produzione!**

Azioni - 3

```
$::RD_AUTOACTION = q{  
  +{ $item[0] => [@item[1..$#item]] };  
}
```

Se la variabile `$::RD_AUTOACTION` è impostata, il suo valore verrà aggiunto ad ogni produzione (a meno che non termini già con una azione!).

Con il codice sopra otterremmo una serie di HASH nidificate aventi per chiave il nome della rule e valore un ARRAYREF contenente il risultato di ogni elemento della produzione...

Azioni - 4

```
{ 'A' => [  
  { 'B' => [  
    'foo',  
    { 'C' => [  
      'bar'  
    ]}  
  ]}  
]}
```

Stiamo quindi iniziando a **elaborare il contenuto**, e non semplicemente ad accettare o meno dati!

<commit>

```
cmd      : proc_cmd | ...
proc_cmd : 'kill' <commit> pid
         | 'ps'
pid      : /\d+/
```

Se viene passato 'kill foo', `proc_cmd` consuma il primo elemento però poi fallisce. E' inutile però continuare con il secondo visto che non puo' iniziare con 'ps'.

La **direttiva** `<commit>` permette appunto di confermare la produzione attuale e impedire al parser di tentare con quelle successive.

<error?>

```
cmd      : proc_cmd | ...
proc_cmd : 'kill' <commit> pid
         | 'ps'
         | <error?>
pid      : /\d+/
```

<error?> (diversamente da <error>) genera un errore solo dopo un <commit>. Nel nostro caso, se arriva un dato che inizia per 'kill' ma non è seguito da un numero:

```
Invalid proc_cmd: Was expecting pid but found
"foo" instead
```

<commit> e <error?>

'kill 3' => OK

'kill foo' => Error!

'ps' => OK

'foobar' => return false

Usare <commit> e <error?> permette di generare **messaggi di errore più precisi** e facilitare così la verifica di errori, sia sui dati che sulla grammatica stessa.

Più in generale, la regola sopra può fallire in 2 modi diversi, generando un errore oppure semplicemente restituendo false.

SQL – 1 (arriviamo al dunque...)

Dimenticandosi (con un senso di liberazione!) dei vari dialetti SQL, prendiamo in considerazione uno schema molto semplificato e cogliamo l'occasione per introdurre le sub-rule:

```
rule : part                # 1 time
rule : part(s)             # 1 or more times
rule : part(s? /,/ /)     # 0 or more times, comma separated
rule : part(?)             # 0 or 1 time
rule : part(s /(?i:AND|OR)/) # 1 or more times,
                             # separed by AND or OR, case insensitive
```

SQL – 2

Cerchiamo ora di costruire una grammatica in grado di accettare:

```
$parser->main('Select foo ,bar from MYTable order BY cuz,bar');
```

Generando un array di un solo elemento:

```
[  
  bless( [ ['foo','bar'], # column(s)  
          'MYTable',    # tablename  
          [],           # where cond  
          ['cuz','bar'], # order by  
        ], 'Select'),  
]
```

SQL – 3 (OMG!)

```
main      : statement(s /;/) /\Z/ { $item[1]; } | <error>
statement : /SELECT/i <commit> column(s /,/ ) from where order_by
          { bless [@item[3,4,5,6]], 'Select' }
          | <error?>
from      : /FROM/i <commit> tablename
          | <error?>
where     : /WHERE/i <commit> <leftop: cond /(AND|OR)/i cond>
          | {[];} # match nothing, return empty ARRAYREF
          | <error?>
order_by  : /ORDER/i <commit> /BY/i column(s /,/ )
          | {[];} | <error?>
tablename : identifier
column    : identifier
identifier: /[a-z][a-z0-9_]*/
```

SQL – 4

```
main      : statement(s /;/) /\z/ { $item[1]; }  
          | <error>
```

Si nota qui l'introduzione di `\z/` usato per il match di fine dati. Questo impone di **consumare tutti i dati** in ingresso, oppure dare un errore.

Inoltre viene restituito solo il primo elemento, che nel caso di una sub-rule è un ARRAYREF.

Siamo quindi non solo in grado di elaborare un singolo statement, ma più di uno separati da punto e virgola.

SQL – 5

```
statement : /SELECT/i <commit> column(s /,/ ) from where order_by  
           { bless [@item[3,4,5,6]], 'Select' }  
           | <error?>
```

- cerca il terminale SELECT, quindi commit.
- cerca di consumare una o più column.
- cerca from, where e order_by.
- restituisce quindi un ARRAYREF blessed 'Select', che contiene i dati **utili** dello statement SQL.

(È facile immaginare un package chiamato Select che esponga metodi di comodo...)

SQL – 6 (<leftop>)

```
where      : /WHERE/i <commit> <leftop: cond /(AND|OR)/i cond>  
          | {[];} # match nothing, return empty ARRAYREF  
          | <error?>
```

La direttiva <leftop> associa a sinistra una sequenza di una o più subrule, in questo caso separate da AND oppure OR.

Restituisce una lista di valori (anche il separatore, se in un gruppo dell'espressione regolare)

La seconda produzione **restituisce invece una lista vuota** nel caso in cui non sia presente la where.

Associatività

Come appena intravisto esiste la direttiva <leftop> (e come prevedibile la <rightop>) che risolvono il problema dell'associatività.

La **sottrazione** è un'operazione aritmetica che **associa a sinistra**.

$$5-4-3 \Rightarrow (5-4)-3 \Rightarrow -2$$

Se invece di associare a sinistra si procedesse da destra si otterrebbe:

$$5-4-3 \Rightarrow 5-(4-3) \Rightarrow 4$$

Evidentemente **sbagliato**.

Associatività - 2

Altri operatori richiedono **associatività a destra**, come l'elevamento a potenza:

$$4^{**}3^{**}2 \Rightarrow 4^{**}(3^{**}2) \Rightarrow 262144 \# \text{ OK}$$

$$4^{**}3^{**}2 \Rightarrow (4^{**}3)^{**}2 \Rightarrow 4096 \# \text{ wrong}$$

Volendo scrivere queste operazioni in una grammatica:

`sum : <leftop: num / (+|-) / num>`

`pow : <rightop: num '**' num>`

Precedenza

Rimanendo in tema di espressioni, va ricordata pure la **precedenza**:

$$7-2*3 \Rightarrow 7-(2*3) \Rightarrow 7-6 \Rightarrow 1$$

Che nel caso di una grammatica non è altro che la profondità di una regola rispetto ad un'altra.

`sum : <leftop: mul / (+|-) / mul>`

`mul : <leftop: pow / (*|\/) / pow>`

`pow : <rightop: num '**' num>`

`num : /-?\d+ /`

Ricorsione

Volendo permettere l'uso di parentesi:

```
sum : <leftop: mul / (+|-) / mul>
```

```
mul : <leftop: pow / (*|\/) / pow>
```

```
pow : <rightop: num '**' num>
```

```
num : /-?\d+/
```

```
| '(' <commit> sum ')'
```

```
| <error?>
```

Dove si nota la ricorsione da num a sum e l'uso di <error?> per le parentesi non chiuse correttamente.

Espressioni – 1

```
#IN> (3+4)*5+6
[# sum          '(3+4)*5+6'
  [# mul        '(3+4)*5'
    [# pow      '(3+4)' ----> simplify!
      #( <commit> '(3+4)'
        [# sum   '3+4'
          [# mul  '3' -----> simplify!
            [# pow '3' -----> simplify!
              3# num '3'
            ]
          ],
          '+',
          [[4]],# mul(pow(4)) ----> simplify!
        ],
        '* ',
        [5],#pow(5) -----> simplify!
      ]
    ],
    '+ ',
    [[6]],#mul(pow(6)) -----> simplify!
  ]
]
```

Espressioni – 2

Possiamo quindi **migliorare** il parser usando una funzione esterna:

```
sum : <leftop: mul / (\+|-) / mul> { ::op(@item) }
mul : <leftop: pow / (\*|\/) / pow> { ::op(@item) }
pow : <rightop: num '**' num> { ::op(@item) }
num : /-?\d+/
    | '(' <commit> sum ')' { $item[3]; }
    | <error?>
```

Espressioni - 2

```
sub op
{
  my (@data) = @_ ;
  if(scalar @data == 2 && ref $data[1] eq 'ARRAY') {
    @data = ($data[0], @{$data[1]})
  }
  return $data[1] unless scalar @data > 2;
  return [@data];
}
```

La funzione sopra collassa i dati in ingresso, così da ottenere risultati tipo **['sum', 3, '-', 2, '+', 5]**, oppure direttamente **'3'** se l'operazione è unaria.

Molto più maneggevoli!

Espressioni – 3 (OMG! - 2)

Con lo stesso ingresso:

$(3+4) * 5 + 6$

Otteniamo il più confortevole:

```
[ 'sum' ,  
  [ 'mul' ,  
    [ 'sum' , '3' , '+' , '4' ] ,  
    '*' ,  
    '5'  
  ] ,  
  '+' ,  
  '6' ,  
];
```

Espressioni – 4 (logica)

Aggiungiamo ora gli operatori logici OR e AND e di comparazione:

```
or   : <leftop: and /OR/i and>      { ::op(@item) }
and  : <leftop: cmp /AND/i cmp>     { ::op(@item) }
cmp  : sum /(<>|<|>|=)/ sum        { ::op(@item) }
sum  : <leftop: mul /(\+|-)/ mul>    { ::op(@item) }
mul  : <leftop: pow /(\*|\/)/ pow>   { ::op(@item) }
pow  : <rightop: num '**' num>      { ::op(@item) }
num  : /-?\d+/
      | '(' <commit> or ')'
      | <error?>
```

Non vedo l'ora di inserire tutto questo dentro il mio parser SQL, e voi?

SQL – 7 (where cond)

Siamo ora in grado di modificare la nostra grammatica per SQL così da includere la logica di condizione:

```
where      : /WHERE/i <commit> where_cond
           | [[];] # match nothing, return empty ARRAYREF
           | <error?>

where_cond : or      # enter the expressions parser
           | column  # allow number or column names
           | '(' <commit> or ')' { $item[3]; }
           | <error?>
```

Ora il nostro parser non solo riconosce le condizioni di where, ma le “interpreta” elaborando le espressioni aritmetiche; e la giusta precedenza perbacco!

SQL – 8 (OMG! - 3)

```
# SELECT foo FROM tab WHERE foo>bar or bar<>3+cuz*2 and cuz/2<foo
[bless( [
  ['foo' ],                                # columns
  'tab',                                   # table
  ['or',                                    # where condition
    ['cmp', 'foo', '>', 'bar' ],           # ...
    ['and',                                    # ...
      ['cmp', 'bar', '<>',
        ['sum', '3', '+',
          ['mul', 'cuz', '*', '2' ]],
      ],
    ],
    ['cmp',
      ['mul', 'cuz', '/', '2' ],
      '<',
      'foo',
    ],
  ],
],
],
[],
], 'Select' ),]
```

SQL – 9

Vediamo infine come permettere l'uso di più tabelle:

```
from      : /FROM/i <commit> from_table(s /,/)  
          | <error?>  
column    : identifier /\./ field { +{@item[1,3]}; }  
          | field                { +{undef, $item[1]}; }  
field     : '*' | identifier  
from_table : tablename /as/i identifier  
          | tablename { +{${item[3]} => $item[1]}; }  
          | tablename { +{${item[1]} => $item[1]}; }
```

Ora il nostro parser cercherà una lista di tabelle, eventualmente con un nickname, e permetterà pure di usare colonne tipo 'alias.field'.

SQL – 10

```
# select a.*, avg from foo as a, bar as b
# where b.id = a.bar and b.avg > a.avg*2
[bless( [
  [{'a' => '*'}, {undef => 'avg'}],           # columns
  [{'a' => 'foo'}, {'b' => 'bar'}],         # tables
  ['and',                                     # where
    ['cmp', {'b' => 'id'}, '=', {'a' => 'bar'}],
    ['cmp', {'b' => 'avg'}, '>',
      ['mul', {'a' => 'avg'}, '*', '2']],
  ],
  []                                           # order by
], 'Select' )]
```

Si nota come la modifica abbia dato riscontri pure sulle condizioni. Inoltre abbiamo permesso di usare avg senza nick di tabella, infatti si vede **undef** come secondo dato dell'array delle colonne di select.

SQL – 11 (<skip>)

Una sequenza di istruzioni select può contenere dei **commenti**, vale a dire del testo preceduto da '--'.

```
SELECT foo, bar, -- a useless comment  
FROM my_table;
```

Per chiedere a P::RD di “**ignorare**” i commenti mentre si elabora il documento in ingresso usiamo <skip>.

Di **default** <skip> è configurato per saltare **spazi, tab e ritorni di carrello**.

SQL – 12 (<skip>)

```
main      : <skip: qr{\s*(--[^\n]*\n\s*)?}s> # skip comments
          : statement(s /;/) # statements
          : /\z/ # end of document
          : { $item[2]; }
          | <error>

statement : /SELECT/i ....
          | {[];} # empty statement
```

Si è così modificato il meccanismo di skip di P::RD per includere anche i commenti.

Abbiamo inoltre permesso di avere statement vuoti, così da permettere punti e virgola extra.

Conclusione

Ovviamente P::RD espone molte più potenzialità di quelle viste finora.

Ed esistono altri tipi di parser, che per esempio anziché partire dall'alto e scendere verso il basso procedono a ritroso.

Obiettivo di questa breve introduzione era mostrare come l'uso di questo modulo possa esser più semplice del previsto e molto produttivo, per il resto vi rimando a CPAN e alle varie guide su P::RD presenti sulla rete.