Journées Perl 2008

(analyse syntaxique selon des principes & méthodes issus de la programmation fonctionnelle)

Le menu de le talk

- **Nous allons parler d'analyse syntaxique**
  - Perl est – à dessein – très adapté à la manipulation textuelle
  - Mais des fois le « *ligne-à-ligne* » n'est plus suffisant
  - Ça arrive dans la vraie vie (et même le CPAN n'y peut rien !)
- **À l'aide d'un algorithme de « descente récursive »**
  - Principes & techniques éculés mais implémentation alternative
  - Proche du « *combinator parsing* » (cf parsec pour Haskell)
- **Au travers d'une approche simple par l'exemple**
  - Pas de « *magie noire* » (du moins au début ☺)
  - Mais plutôt essayer d'assimiler correctement les bases
  - Principes, techniques puis application sur un exemple concret

Préambule

(analyse syntaxique selon des principes & méthodes issus de la programmation fonctionnelle)

Le HOPi sans peine ?

- **Tout a commencé…**
  - Par un livre que jamais il ne comprît !
  - Découvert par hasard sur la liste des Mongueurs de Perl
- **« *Higher Order Perl* » de Mark-Jason Dominus**
  - En particulier le chapitre 8 (p376-p415, 1ére édition)
- **Principes/techniques de ProgrammaΘ Fonctionnelle**
  - Usage intensif de fermetures (« *closures* »)
  - Listes infinies, évaluation « *fainéante* » (« *lazy* »)
  - Fonctions transformant d'autres fonctions (citoyennes de 1ère cl.)
- **Excellent livre**
  - Mais l'aspirine n'est pas livrée avec !

Pourquoi analyser ?

- **Vérification syntaxique**
  - Le flux est-il conforme à la grammaire ?
  - KO si jeton (lexer) ou séquence de jetons (parser) non reconnus
- **Génération d'une structure**
  - Par défaut très proche de la grammaire
  - Mais il est possible de l'adapter à la volée
- **Plusieurs types de structures en sortie**
  - Liste de listes (~ transformation)
  - Arbitraire par effet de bord sur une variable d'état (~ parallèle)
  - Évaluation à la volée (~ réduction)

Pourquoi HOP ?

- **Système ouvert**
  - Par opposition à d'autres, excellents mais fermés
- **Basé sur des principes simples**
  - Recherche, concaténation, alternance & transformation
- **Élégant ?**
  - Avec la syntaxe ad'hoc ! (mon cuisant échec avec `overload`...)
  - Le parser est du *code* et non pas des *données* (e.g., YACC/P::RD)
- **Problèmes**
  - Fuites de mémoire (fermetures) en fonction de la version de Perl
  - Difficile à déverminer (HOP::* à améliorer)
  - Ticket d'entrée plutôt élevé (comme dans tout système récursif ?)

HOP en action

- **Déjà sur le CPAN**
  - Modules HOP::* maintenus par Curtis « Ovid » Poe
  - Code recopié quasi-verbatim
  - Mais il faut quand même avoir le livre (POD sub-optimal)
  - Le livre sera néanmoins un jour disponible sur le ouaibe
- **HOP::Stream**
  - Listes chaînées potentiellement infinies (notion de « *promesse* »)
- **HOP::Lexer**
  - Agrégation en valeurs typées (identification par regexp)
- **HOP::Parser**
  - Assembler récursivement un parser à l'aide de plus simples

Principes

(analyse syntaxique selon des principes & méthodes issus de la programmation fonctionnelle)

Analyse syntaxique [1]

- **Définition**
  - Structure – si possible – un flux non-structuré
  - L'entrée est souvent un flux de caractères
  - La sortie est souvent une structure arborescente (si OK)
  - Ça passe ou ça casse…
- **Décomposée en deux phases distinctes**
  - L'analyse lexicale (en Anglais « *lexer* »)
  - L'analyse grammaticale (en Anglais « *parser* »)
  - Pour chaque étape, ça passe ou ça casse…
- **En Anglais, « *to parse* »**
  - Dans ce qui suit : « *parser* » (non-Gaulois mais court ☺)

Analyse syntaxique [2]

- **Phase 1 : Analyse lexicale**
  - Découpe le flux incident de caractères en une séquence de mots
  - Fournit une liste de jetons (« *tokens* ») à la phase suivante
  - Un jeton peut déjà être constitué par : `[ $type => $valeur ]`
  - Les agrégats de caractères peuvent être « *typés* » à la volée
- **Phase 2 : Analyse grammaticale**
  - Essaye de construire des phrases à partir de la séquence de mots
  - Tente de faire correspondre une grammaire formelle
  - La grammaire est définie à l'aide de la notation EBNF
  - Retourne une structure attendue par l'application

Analyse lexicale

- **Procède caractère par caractère**
  - Agrégation de séquences de caractères en mots
- **Caractères ? Mots ? Regexp !**
  - Les regexps de Perl possèdent des fonctions dédiées
  - Le méta-caractère \G avec les options /gc (cf perlre)
  - Se souvient en fait de la position de la dernière correspondance
  - Dans le cadre d'une utilisation incrémentale
- **HOP::Lexer**
  - Basé sur HOP::Stream (listes infinies car « *paresseuses* »)
  - En Français : « *flux* »
  - Construit un lexer à l'aide d'une liste de regexps

Analyse grammaticale

- **Grammaire formelle**
  - Par exemple EBNF (« *Extended Backus-Naur Form* »)
  - Meta-langage non-ambigu (car prouvé mathématiquement)
- **EBNF**
  - productions, symboles (non-terminaux) & terminaux
  -
```
FOO := BAR | BAZ          # choix entre symboles
BAR := ( GA BU )?         # séquence optionnelle de terminaux
BAZ := ZO MEU+            # ? ~ un ou plus
GA  := '0'                # terminal
BU  := '1'                # terminal
ZO  := '2'                # terminal
MEU := '3'                # terminal
```
  - «», « *0 1* », « *0 1 0 1* », « *2 3* », « *2 3 3 3 3 3* », etc.
- **Plus à ce sujet sur Wikipedia**

Techniques

(analyse syntaxique selon des principes & méthodes issus de la programmation fonctionnelle)

Intendance

- **Il faut une syntaxe au service de cette sémantique**
  - Perl la propose, « *TIMTOWTDI* » oblige
- **Les fonctions sont citoyennes de 1ère classe**
  - ```
    my $foo = make_foo (); # création
    my $bar = $foo->();    # invocation
    $baz->( $foo );        # paramètre
    ```
- **Support des fermetures « *closures* »**
  - ```
    sub make_foo {
      my $foo = 0; # $foo, locale, sera « enfermée »

      return sub { # renvoie une fonction + ses dépendances
        $foo++;
      };
    }
    ```

Fondamentaux

- **Flux + grammaire**
  - Les parsers travaillent sur un *flux* de *jetons* issu d'un lexer
  - Ce flux est comparé à l'implémentation d'une *grammaire* formelle
- **Recherche (de jeton)**
  - En tête de flux, correspondance par type ou valeur
- **Séquence (de jetons)**
  - Concaténation de plusieurs *recherches* (OK si tous le sont)
- **Alternatives (entre jetons)**
  - Choix entre plusieurs recherches (OK si la première l'est)
- **Transformation**
  - Affecte la *valeur de retour* (fonctionnel mais non structurel)

Lexer (!HOP) [1]

```
sub tokens { # lexer pour une équation (e.g., " !( a&!b | !a&b )")
  my $s = shift;

  my $lexer = sub {
    TOP: {
      return                          if $s =~ m/\G \z            /gxc; # end of string

      return [ CONSTANT => $1 ]       if $s =~ m/\G (\d+)         /gxc;
      return [ VARIABLE => $1 ]       if $s =~ m/\G ([A-Za-z]\w*) /gxc;
      return [ UNARY    => $1 ]       if $s =~ m/\G (\!)          /gxc;
      return [ BINARY   => $1 ]       if $s =~ m/\G ([&|^])       /gxc;
      return [ BRACKET  => $1 ]       if $s =~ m/\G ([\(\)])      /gxc;

      redo   TOP                      if $s =~ m/\G \s+           /gxc; # ignore space(s)

      die "Unknown '$1' char!"        if $s =~ m/\G (.)           /gxc;
    }
  };

  my @tokens;

  while ( my $token = $lexer->() ) {
    push @tokens, $token;
  }

  return @tokens;
}
```

assertion

modificateur

type+valeur

non-conforme

Lexer (HOP) [2]

```
use HOP::Lexer qw( :all );

use constant LEXEMES => (
  [ CONSTANT    => qr/ \d+              /x                ],
  [ VARIABLE    => qr/ [A-Za-z]\w*      /x                ],
  [ UNARY       => qr/ \!               /x                ],
  [ BINARY      => qr/ [&|^]            /x                ],
  [ BRACKET     => qr/ [\(\)]           /x                ],
  [ WHITESPACE  => qr/ \s+              /x  => sub { '' } ],
);

sub lexer {
  my $s = shift;

  my $lexer = string_lexer ( $s, LEXEMES );

  my @tokens;

  while ( my $token = $lexer->() ) {
    push @tokens, $token;
  }

  return @tokens;
}
```

Parser (atomique)

- **La brique de base de tout le système**
  - Destinée à être composée à volonté
- **Très simple**
  - Cherche un jeton donné
  - Comme premier jeton d'un flux de jetons
- **Si OK**
  - Retourne la valeur du jeton
  - Consomme le jeton (i.e., décapite le flux de jetons)
- **Si KO**
  - Laisse la liste de jetons intacte

Parser (composite)

- **Construit à l'aide d'autres parsers**
  - Atomiques ou non
- **Récursivement composés en de plus complexes**
  - Séquence de parsers
  - Alternance de parsers
- **L'un d'entre eux sera le parser applicatif**
- **Ça ressemble pas mal aux regexps, non ?**
  - Les jetons sont les classes de caractères
  - Les séquences sont représentées telles qu'elles
  - L'alternance est notée '|'
  - La structure de sortie est une liste de sous-chaînes

Implémentation [1]

- **HOP::Parser::lookfor (~ recherche)**
  -
```
my $parser = lookfor (   $type              );       # simplifié (ou)
my $parser = lookfor ( [ $type, @valeurs ] );       # simplifié

my ( $parsed, $flux_restant ) = $parser->( $flux ); # motif générique
```
- **Deux parties distinctes constituent le retour**
  - Les valeurs issues de ce qui a déjà été OK : `$parsed`
  - Le reste du flux à parser : `$flux_restant`
- **Ça passe ou ça casse ?**
  - C'était en effet une des motivations pour écrire un parser
  - Que le parser soit atomique ou composite, c'est OK si :
  - `  defined $parsed        # par défaut, liste de listes`
  - `! defined $flux_restant # tout a été consommé`

Implémentation 2

- **Le parser est du code et non point des données**
  - À l'opposé d'autres techniques basées sur les mêmes principes
  - Ça rend HOP particulièrement flexible
- **HOP::Parser::optional**
  -
```
sub optional {              # Alter-ego de '?' pour les regexps
  my $parser = shift;       # agit sur un parser (« Higher Order »)
  return alternate (        # en le composant via un choix :
    $parser,                #   - soit le parser incident est OK
    \&null_list,            #   - soit on essaye autre chose
  );
}

sub null_list {
  my $input = shift;
  return ( [], $input ); # toujours OK & retourne le flux intact
}
```

Implémentation 3

- **HOP::Parser::concatenate**

```
sub concatenate {
  my @parsers = @_;                          # séquence de parsers

  return \&nothing   if @parsers == 0; # garde-fou
  return $parsers[0] if @parsers == 1; # garde-fou

  return sub {                               # créé bien un parser
    my $input = shift;                       # flux, entrée du nouveau parser

    my ( $v, @values );

    for ( @parsers ) {                       # attention, pas de backtracking
      ( $v, $input ) = $_->( $input ); # droite décapité -> gauche
      push @values, $v if defined $v;  # accumule les valeurs
    }

    return ( \@values, $input );             # motif récurrent
  };
}
```

Application

(analyse syntaxique selon des principes & méthodes issus de la programmation fonctionnelle)

Le (mauvais) prétexte

- **Vérification croisée de 2 modèles**
  - L'un – comportemental – en Verilog HDL (norme IEEE)
  - L'autre – statique – en XML (schéma « *maison* »)
- **Modèle comportemental**
  - Verilog a une syntaxe alambiquée mais normée !
  - Les fichiers a traiter sont construits à la main
  - Le lexer est Verilog::Perl (CPAN)
- **Modèle statique**
  - Fichiers générés par un outil « *maison* » dédié
  - Mais attention au principe GIGO !
  - Traitement de type DOM à l'aide de XML::LibXML (CPAN)

Exemple de verilog

- **Bloc « specify »**

```
module dfcrq1 (q, d, cp, cdn);
  output q;
  input cp;
  input d;
  input cdn;

  reg notifier;

  buf(cdn_eq_1, cdn);

  UDP_DFF ( q, d, cp, 1'b1, cdn, notifier );

  specify
    if (cdn)
      (posedge cp => (q +: d)) = (0, 0);
    (negedge cdn => (q +: 1'b0)) = (0, 0);

    $setuphold(posedge cp &&& cdn_eq_1, posedge d, 0, 0, notifier);
    $setuphold(posedge cp &&& cdn_eq_1, negedge d, 0, 0, notifier);
    $recovery(posedge cdn, posedge cp, 0, notifier);
    $hold(posedge cp, posedge cdn, 0, notifier);
    $width(negedge cdn, 0, 0, notifier);
    $width(negedge cp &&& cdn_eq_1, 0, 0, notifier);
    $width(posedge cp &&& cdn_eq_1, 0, 0, notifier);
  endspecify
endmodule
```

Grammaire (EBNF)

- **Extrait du LRM IEEE 1364-2001**

```
$setuphold_timing_check ::= (From Annex A – A.7.5.1)
$setuphold ( reference_event , data_event , timing_check_limit , timing_check_limit
  [ , [ notify_reg ] [ , [ stamptime_condition ] [ , [ checktime_condition ]
  [ , [ delayed_reference ] [ , [ delayed_data ] ] ] ] ] ] ) ;
checktime_condition ::= (From Annex A – A.7.5.2)
  mintypmax_expression
data_event ::=
  timing_check_event
delayed_data ::=
  terminal_identifier
| terminal_identifier [ constant_mintypmax_expression ]
delayed_reference ::=
  terminal_identifier
| terminal_identifier [ constant_mintypmax_expression ]
notify_reg ::=
  variable_identifier
reference_event ::=
  timing_check_event
stamptime_condition ::=
  mintypmax_expression
timing_check_limit ::=
  expression
```

séquence

option

choix

- **Définition récursive (séquence, choix & option)**

Lexer

- **Verilog::Perl a un parser de type flux**
  - Mais pas de « *flux* » au sens HOP
  - Sous-classe et surcharge de « callbacks » (très classique)
- **Décompose en jetons typés à la volée**
  - Keyword, Number, Operator, Symbol
- **Transformer la liste en flux**
  - Simplement à l'aide de `HOP::Stream::list_to_stream ()`

Lexer → Parser

- **En sortie de Verilog::Perl**
  - `$setuphold ( posedge cp &&& cdn_eq_1 , posedge d , 0 , 0 , notifier ) ;`
  -
```
S: $setuphold
O: (
K: posedge
S: cp
O: &&&
S: cdn_eq_1
O: ,
K: posedge
S: d
O: ,
N: 0
O: ,
N: 0
O: ,
S: notifier
O: )
O: ;
```
- **Un embryon de typage**
  - S=Symbol, O=Operator, K=Keyword, N=Number

Parser (code) [1]

```
my $and_triple         = lookfor ( [ O => '&&&' ] ); # type + valeur
my $comma              = lookfor ( [ O => ','   ] );
my $open               = lookfor ( [ O => '('   ] );
my $close              = lookfor ( [ O => ')'   ] );
my $semicolon          = lookfor ( [ O => ';'   ] );

my $number             = lookfor ( [ 'N' ] );        # type seulement
my $symbol             = lookfor ( [ 'S' ] );

my $symbol_or_number   = alternate ( $symbol, $number );

my $task_sh_rr         = alternate ( lookfor ( [ S => '$setuphold' ] ), lookfor ( [ S => '$recrem' ] ) );
my $task_s_h_r_r       = alternate ( lookfor ( [ S => '$setup'    ] ), lookfor ( [ S => '$hold'    ] ),
                                     lookfor ( [ S => '$recovery' ] ), lookfor ( [ S => '$removal' ] ) );
my $task_width         = lookfor ( [ S => '$width' ] );

my $edge               = alternate ( lookfor ( [ K => 'posedge' ] ), lookfor ( [ K => 'negedge' ] ) );

my $optional_number    = optional ( concatenate ( $comma => $number ) );
my $optional_symbol    = optional ( concatenate ( $comma => $symbol ) );
my $optional_condition = optional ( concatenate ( $and_triple, $symbol ) );

my $edged_symbol       = concatenate ( $edge, $symbol, $optional_condition );
```

Parser (code) ²

```
my $sh_rr = concatenate (
  $task_sh_rr, $open,
    $edged_symbol,                  $comma,   # reference event
    $edged_symbol,                  $comma,   # data      event
    $symbol_or_number,              $comma,   # setup     limit
    $symbol_or_number,              $comma,   # hold      limit
    list_of ( optional ( $symbol ), $comma ), # [notifier, timestamp/timecheck cond, delayed ref./data]
  $close, $semicolon
);
my $s_h_r_r = concatenate (
  $task_s_h_r_r, $open,
    $edged_symbol, $comma,                    # data      (setup) or reference (other) event
    $edged_symbol, $comma,                    # reference (setup) or data      (other) event
    $symbol_or_number,                        # limit
    $optional_symbol,                         # [notifier]
  $close, $semicolon
);
my $width = concatenate (
  $task_width, $open,
    $edged_symbol, $comma,                    # reference event
    $symbol_or_number,                        # limit
    $optional_number,                         # [threshold]
    $optional_symbol,                         # [notifier]
  $close, $semicolon
);

my $parser = alternate ( $width, $sh_rr, $s_h_r_r );

my @return;
eval { @return = $parser->( list_to_stream ( @_ ) ) };
return undef if $@;
return @return;
```

Sortie brute

- **En sortie du parser**

```
$returned = [
  [                                                  # my $sh_rr = concatenate ( ... )
    '$setuphold',                                    # alternate ( lookfor ( [ S => '$setuphold' ] ), ...
    '(',                                             # lookfor ( [ O => '(' ] );
    [ 'posedge', 'cp', [ '&&&', 'cdn_eq_1' ] ], # concatenate ( $edge, $symbol, $optional_condition )
    ',',
    [ 'posedge', 'd',  []                      ], # optional ( concatenate ( $and_triple, $symbol ) )
    ',',
    '0',                                             # alternate ( $symbol, $number )
    ',',
    '0',
    ',',
    [ [ 'notifier' ] ],                              # list_of ( optional ( $symbol ), $comma )
    ')',
    ';'
  ],
  undef                                              # pas de reste
];
```

- **Remarques**
  - « `concatenate` » génère un niveau de liste supplémentaire
  - « `list_of` » dégage le séparateur

Bidouille Grenouille [1]

Brut

```
$returned = [
  [
    '$setuphold',
    '(',
    [ 'posedge', 'cp', [ '&&&', 'cdn_eq_1' ] ],
    ',',
    [ 'posedge', 'd',  []                    ],
    ',',
    '0',
    ',',
    '0',
    ',',
    [ [ 'notifier' ] ],
    ')',
    ';'
  ],
  undef
];
```

Attendu

```
$returned = [
  [
    '$setuphold',
                                            # ❶
    [ 'posedge', 'cp', 'cdn_eq_1' ], # ❷
                                            # ❶
    [ 'posedge', 'd'              ], # ❸
                                            # ❶
                                            # ❹
                                            # ❶
                                            # ❹
                                            # ❶
                                            # ❹
                                            # ❶
                                            # ❶
  ],
  undef,
];
```

Bidouilles

- ❶ : enlever la ponctuation
- ❷ : réduire le niveau de [ ]

...

- ❸ : simplifier (undef vs [ ])
- ❹ : éliminer le superflu

Bidouille Grenouille [2]

- ❷ : réduire le niveau de []
  ❸ : simplifier (undef vs [])
  - `concatenate ( $edge, $symbol, $optional_condition )`
  - `T ( concatenate ( $edge, $symbol, $optional_condition ), sub { $_[2][0] ? [ $_[0], $_[1], $_[2][1] ] : [ $_[0], $_[1] ] } )`
  - ❷ ❸
  - `$optional_condition = optional ( concatenate ( $and_triple, $symbol ) );`
  - Concatenate ❷ `$_[2][1]`
- T() – opérateur de transformation
  - `my $nouveau_parser = T ( $parser, $sub )`
  - Retourne un *parser* appliquant $sub au *résultat* de $parser
  - Les paramètres respectent l'ordre dans concatenate ()

Bidouille Grenouille 3

- ❶ : enlever la ponctuation
  ❹ : éliminer le superflu
  - `alternate ( $width, $sh_rr, $s_h_r_r )`
  - `T ( alternate ( $width, $sh_rr, $s_h_r_r ), sub { [ $_[0], $_[2], $_[4] ] } )`
  -
```
$sh_rr = concatenate (
  $task_sh_rr, $open,                          # [0], [1]
    $edged_symbol,                  $comma,    # [2], [3]
    $edged_symbol,                  $comma,    # [4], [5]
    $symbol_or_number,              $comma,    # [6], [7]
    $symbol_or_number,              $comma,    # [8], [9]
    list_of ( optional ( $symbol ), $comma ), # …
  $close, $semicolon
);
```

Limites

(analyse syntaxique selon des principes & méthodes issus de la programmation fonctionnelle)

Limites

- **Pas implémenté dans HOP::Parser**
  - À considérer encore comme du code en état alpha (dixit Curtis)
  - Mais c'est tout à fait abordé et expliqué dans le bouquin
- **Backtracking**
  - En pratique, il est souvent possible de le contourner
  - Avec des ruses du genre « *sommes de facteurs* »
- **Debug**
  - Avoir le nom et le niveau de profondeur d'un parser
  - Connaître le statut d'un parser (réussite ou échec)
- **Documentation**
  - « *Ne souffrirait pas de quelques améliorations* »

Épilogue

(analyse syntaxique selon des principes & méthodes issus de la programmation fonctionnelle)

Q&A

Remerciements