Mêler combinateurs, continuations et *EBNF* pour une analyse syntaxique efficace en `OCaml`

Rodolphe Lepigre et Christophe Raffalli

LAMA, UMR 5127 CNRS, Université Savoie Mont Blanc
73376 Le Bourget-du-Lac CEDEX, France
rodolphe.lepigre@univ-savoie.fr
christophe.raffalli@univ-savoie.fr

Résumé

Le cœur de `DeCaP`, objet de cet article, est une bibliothèque de combinateurs d'analyseurs syntaxiques. Ils sont la cible de la traduction d'une syntaxe *EBNF*, sans récursion à gauche, que l'on a ajoutée à `OCaml`. Les parseurs ainsi définis sont des expressions de première classe.

Pour plus d'efficacité, nos combinateurs utilisent des continuations et inspectent l'ensemble des premiers caractères acceptés par une grammaire afin d'élaguer l'arbre des possibilités. Les continuations donnent naturellement la sémantique complète d'*EBNF* et `DeCaP` peut donc gérer les grammaires ambigües. De plus, des combinateurs inspirés de la notion de continuation délimitée permettent d'optimiser certaines grammaires en restreignant la sémantique. Notre parseur d'`OCaml` est en moyenne deux fois plus rapide que celui de `Camlp4` et cinq fois plus lent que l'original.

`DeCaP` fournit également un système de *quotation* et d'*anti-quotation* similaire à celui de `Camlp4` et permet ainsi d'étendre la syntaxe d'`OCaml`. Notre outil se veut plus simple et moins contraignant que ce dernier et n'impose, par exemple, aucune analyse lexicale.

Introduction

Ce travail est un sous-produit du développement de `Patoline` [8], un nouveau système de composition de documents initié en 2011 par Pierre-Étienne Meunier. La syntaxe de `Patoline` mélange la grammaire d'`OCaml` et une variante de celle de LaTeX. On peut donc écrire du `OCaml` en `Patoline` et vice-versa. La première version du langage, utilisant `Dypgen` [9] et `Camlp4` [2], n'est pas suffisamment robuste. En particulier, La gestion de la position des erreurs à l'interface des deux grammaires est problématique. L'utilisation de `Camlp4` seul est impossible car l'analyse lexicale d'`OCaml` ne convient pas à une grammaire de style LaTeX.

Nous avons finalement décidé d'écrire un outil similaire à `Camlp4` n'utilisant aucune analyse lexicale. Certaines idées présentes dans `Dypgen` ont été retenues, comme la gestion intégrée des caractères non-significatifs.

Dans un cadre plus général, les langages spécifiques (DSL) se multiplient : formats d'échange standards, spécifications de protocoles, fichiers de configuration... Il est donc souhaitable de disposer d'outils à la fois simples d'utilisation et le plus expressifs possible. Le traitement informatique des langues naturelles se développe aussi et nécessite des outils permettant de travailler avec des grammaires ambigües.

Est-il possible de créer un seul outil satisfaisant tous ces besoins dans le cadre d'un langage fonctionnel ? Examinons d'abord les solutions existantes classifiées en deux catégories : les systèmes à la *BNF* (forme de Backus-Naur) et les combinateurs. Les premiers proposent d'écrire le langage

visé dans un format proche de *BNF* ou d'*EBNF*, compilé dans la plupart des cas vers des automates à pile, réputés plus efficace que les seconds. L'utilisation de *BNF* permet en outre de générer une documentation du langage.

***BNF* Converter [10]**, proposé par Forsberg, Pelauer et Ranta permet de générer, en plus d'un lexeur et d'un parseur, un type de données représentant l'arbre de syntaxe abstraite du langage, équipé d'une fonction de parcours et d'un système d'impression élégante. À partir d'un fichier commun rédigé en syntaxe *Labeled BNF*, *BNFC* permet de générer l'implantation du parseur correspondant dans divers langages dont `OCaml`, `Haskell`, `C` ou `Java`.

`OCamlYacc`, `Menhir` [11] et `Happy` [7] : le premier est une modification du code `C` de `Yacc` pour générer du code `OCaml`. Le second, proposé par François Pottier et Yann Régis-Gianas, est écrit en `OCaml` et est en grande partie compatible avec `OCamlYacc`. Il y a cependant quelques différences notables entre ces deux outils : `Menhir` produit des analyseurs réentrants LR(1), alors qu'`OCamlYacc` produit des analyseurs LALR(1) non réentrants. Les messages d'erreurs de `Menhir` se réfèrent à la grammaire et non à l'automate. `Happy` [7], écrit par Marlow et Gill, est un outil similaire à `Menhir` pour le langage `Haskell`.

Camlp4 [2], écrit par Daniel de Rauglaudre, permet d'écrire des analyseurs de syntaxe directement en `OCaml` et d'en étendre la syntaxe. Il est toutefois difficile à utiliser et, en plus de dix ans, il ne s'est pas imposé comme *la* solution pour écrire des langages en `OCaml`.

Dypgen [9], développé à partir de 2005 par Emmanuel Onzon, est un générateur de parseurs GLR qui met l'accent sur les grammaires auto-extensibles. Dypgen permet de définir en même temps le lexeur et le parseur en gérant les ambigüités du lexeur et propose un système de contrôle des caractères blancs. Ces innovations lui confèrent un supplément d'expressivité.

Les bibliothèques de combinateurs, quant à elles, permettent d'écrire et de composer des parseurs atomiques, afin d'en former de nouveaux. On tire ainsi parti des structures d'ordre supérieur proposées par les langages fonctionnels pour manipuler les parseurs comme des valeurs de première classe. On profite alors des garanties offertes par le langage hôte (typage, réentrance, ...).

`Parsec` [5] est généralement acceptée comme étant la première bibliothèque de combinateurs utilisable en pratique. Implantée par Daan Leijen et Erik Meijer pour le langage `Haskell`, elle offre deux améliorations notables vis-à-vis de ses prédécesseurs. Les messages d'erreurs générés sont plus précis et contiennent la position de l'erreur ainsi que la liste des terminaux qui auraient pu être acceptés à cette position. L'implantation est également plus efficace en temps et en espace que les implantations naïves précédentes. Il est important de noter que, comme la plupart des bibliothèques de combinateurs, `Parsec` ne peut pas être utilisé pour écrire des grammaires récursives à gauche.

`Planck` [3] a été écrit sans connaître `Parsec`, avec pour seul objectif d'obtenir une bibliothèque de combinateurs monadiques. L'auteur a constaté rétrospectivement une certaine similarité avec `Parsec`. Ceci semble montrer que l'approche des combinateurs est naturelle, tant au niveau de la conception que de l'implantation. `Planck` a été utilisé pour écrire un parseur complet pour `OCaml`, démontrant qu'un tel outil est utilisable en pratique, même si ses performances sont un peu décevantes (70 à 100 fois plus lent que le parseur écrit avec `OCamlYacc`).

`Ostap` [1] est une bibliothèque de combinateurs de parseurs qui ne se veut pas originale. Une extension de syntaxe, écrite en `Camlp4`, offre la possibilité de spécifier des grammaires *EBNF* qui sont traduites vers des expressions à base de combinateurs. Cette transformation préserve le caractère première classe des parseurs. `Ostap` a également été utilisé pour écrire un parseur complet pour `OCaml`.

Pourquoi un autre outil ? Aucune des solutions énumérées ci-dessus ne satisfait completement les besoins de `Patoline`. Nous proposons donc un nouvel outil, que nous voulons le plus général possible, reprenant beaucoup d'idées des logiciels précédemment cités : gestion des grammaires ambigües et des espacements de `Dypgen`, combinateurs offrant des parseurs de première classe, syntaxe *EBNF* comme `Ostap`, mécanisme d'extension comme `Camlp4`. Nous essayons également de nous approcher

au maximum de l'efficacité d'`OCamlYacc`.

Nous avons choisi d'utiliser des continuations au sein de `DeCaP`, pour des raisons d'efficacités [12, 4]. Par contre, dans le cas de `Parsec` [5], l'auteur présente les continuations comme moins efficaces. En fait, comme nous le verrons, les combinateurs naturels sont plus généraux avec des continuations. La comparaison est donc délicate. Cette approche a un autre intérêt : comme remarqué par Koopman et Plasmeijer [4], l'utilisation de continuations délimitées permet de contrôler finement la sémantique des combinateurs et l'ambigüité des grammaires. `DeCaP` offre ainsi la possibilité de générer tous les arbres de syntaxe possibles dans le cas d'une grammaire ambigüe.

Les continuations seules ne suffisent pas pour atteindre l'efficacité voulue. Koopman et Plasmeijer [4] proposent d'utiliser le combinateur d'alternatives exclusives d'Haskell. Nous proposons une autre solution : utiliser une monade plus complexe, stockant des informations supplémentaires qui permettent d'éliminer certaines alternatives.

Nous pensons que les continuations peuvent également permettre de lever la limitation concernant la récursion à gauche. Nous avons fait quelques avancées dans cette direction que nous relatons ici, même si nous n'en avons rien gardées dans l'implantation actuelle.

`DeCaP`, comme beaucoup de bibliothèques de combinateurs, offre l'avantage de s'utiliser sans analyseur lexical. Cela a pour conséquence d'alourdir la définition des grammaires. Pour résoudre ce problème on intègre à la monade une notion de fonction *blank*, qui est utilisée automatiquement après chaque terminal de la grammaire pour ignorer les caractères non significatifs (espaces, commentaires...). Il est aussi possible de changer ce comportement en cours d'analyse syntaxique, ce qui est indispensable pour les grammaires hétérogènes, comme celle de `Patoline` [8].

1. Syntaxe *EBNF*, intégration à `OCaml`

Nous commençons par donner une idée générale de `DeCaP`, à travers des exemples. On en profitera pour clarifier la syntaxe *EBNF* utilisée par `DeCaP`, qui sera également utilisée dans le reste de cet article pour spécifier des grammaires. Le manuel utilisateur de `DeCaP` [6] contient davantage d'exemples et donne plus d'explications vis-à-vis de la syntaxe *EBNF* et des combinateurs en général.

Premier exemple : un grand classique

Le listing 1 contient l'implantation d'une calculatrice avec sommes, produits et puissances. Cette grammaire dépend d'un paramètre `prio` afin de respecter les règles de priorités habituelles. Une telle grammaire récursive doit être déclarée au préalable en utilisant la fonction `grammar_family`, qui produit un couple contenant la grammaire, ainsi qu'une fonction permettant de la définir (ici `set_expr`). Le mot-clé `parser` permet ensuite de donner les règles de grammaire en utilisant un format de type *EBNF*. Afin d'assurer le respect des priorités, les alternatives de la grammaire sont gardées par une clause `when`[1]. La sémantique d'une règle est donnée à droite de la flèche, et il est possible de définir des règles locales en utilisant des accolades.

Pour mieux comprendre cette grammaire, voici la liste des combinateurs terminaux de `DeCaP` :

`ANY` parse un caractère quelconque différent du caractère de fin de fichier. La valeur sémantique de ce parseur est le caractère lui même.

`CHR(c)` parse uniquement le caractère `c`. On peut utiliser directement `'x'` à la place de `CHR('x')`.

`STR(s)` similaire au précédent pour une chaîne avec la même convention pour les constantes.

1. Il faut noter que les clauses `when` ne peuvent pas faire référence aux noms des terminaux, mais uniquement aux variables accessibles au niveau du mot-clé `parser`. Ces clauses sont en effet testées avant d'analyser le flux. On peut lever l'exception `Give_up message` dans la sémantique pour faire échouer l'analyse.

Listing 1 – la calculatrice

```
type prio = Sum | Prod | Pow | Atom

let expr, set_expr = Decap.grammar_family "expr"
let _ = set_expr (fun prio ->
  parser
  | f:''[0-9]+\([.][0-9]+\)?'' when prio = Atom -> float_of_string f
  | "(" e:(expr Sum) ")"       when prio = Atom -> e
  | e:(expr Atom) l:{"**" (expr Atom)}*
                               when prio = Pow  -> List.fold_left ( ** ) e l
  | e:(expr Pow)  l:{"*"  (expr Pow)}*
                               when prio = Prod -> List.fold_left ( *. ) e l
  | e:(expr Prod) l:{"+"  (expr Prod)}*
                               when prio = Sum  -> List.fold_left ( +. ) e l)
let _ =
  let blank = Decap.blank_regexp ''[ \t]*'' in
  try while true do
    Printf.printf ">> %!";
    let l = input_line stdin in
    let r = Decap.handle_exception (Decap.parse_string (expr Sum) blank) l in
    Printf.printf "%f\n%!" r
  done with End_of_file -> ()
```

`RE(r)` parse le flux d'entrée selon l'expression régulière `r`, et retourne la chaîne parsée. La syntaxe `''regexp''` est équivalente à `RE("regexp")` sauf qu'il n'est pas nécessaire d'échapper « \ ».

`EOF` parse le caractère de fin de fichier et a pour valeur sémantique () : `unit`.

`EMPTY` ne parse rien et a pour valeur sémantique ().

`FAIL(msg)` échoue directement, en ajoutant `msg` à la liste des messages d'erreur.

`DEBUG(msg)` ne parse rien, mais affiche des informations de déboguage ainsi que le message `msg`.

Comme nous pouvons le voir dans l'exemple de la calculatrice (listing 1), la mise en séquence s'effectue par simple juxtaposition après le mot-clé `parser`. Lorsque qu'aucune action n'est spécifiée, la valeur sémantique retournée correspond au tuple des valeurs des éléments de la séquence, hormis les terminaux comme `CHR` ou `STR` [2]. On peut également grouper les terminaux en utilisant des accolades. {...} est une abréviation pour (`parser`...) à l'intérieur d'une règle. Cela permet d'utiliser un parseur sans le nommer. Les opérateurs à la Kleene (« ∗ », « + » et « ? ») modifient la sémantique de la manière attendue avec les types `list` et `option` respectivement.

On utilisera par la suite la syntaxe de `DeCaP` avec accolades pour définir des langages. La grammaire acceptant un nombre arbitraire de caractères « `a` » suivi d'un nombre arbitraire de caractères « `b` » ou « `c` » sera donc notée `{'a'* {'b'|'c'}*}`.

Extensions de syntaxe, *quotations* et *anti-quotations*

En plus de l'écriture d'analyseurs syntaxiques, `DeCaP` propose un parseur extensible pour `OCaml`. Le mécanisme s'inspire beaucoup de `Camlp4` en tentant d'être plus flexible et plus simple à utiliser. En particulier, les extensions sont des foncteurs que l'on compose pour obtenir la grammaire finale. On a repris aussi les notions de *quotation* et d'*anti-quotation*. `DeCaP` fournit par ailleurs un préprocesseur qui permet, par exemple, de compiler du code en fonction de la version d'`OCaml`. Le listing 2 présente un exemple ajoutant des expressions `do` ... `where` et `let try` ... à `OCaml`.

2. On peut ignorer certaines valeurs en utilisant la notation `_:g`.

Listing 2 – exemple d'extension de syntaxe

```
module Ext = functor(In:Extension) -> struct
  include In

  let extension = parser
  | STR("do") e:(expr) STR("where") r:STR("rec")? b:let_binding ->
      (Let, if r<>None then <:expr<let rec $bindings:b$ in $e$>>
            else           <:expr<let $bindings:b$ in $e$>>)

  | STR("let") STR("try") b:let_binding
    STR("in") e:(expr) STR("with") c:(match_cases Top) ->
       let c = List.map (fun (pat, e) -> (pat, <:expr< fun () -> $e$ >>)) c in
       (Let, <:expr<(try let $bindings:b$ in fun () -> $e$ with $cases:c$) ()>>)

  let extra_expressions = extension::extra_expressions
  let _ = add_reserved_id "where"
end
```

2. La librairie de combinateurs de `DeCaP`

`DeCaP` est une bibliothèque de combinateurs de parseurs monadiques pour le langage `OCaml`, avec des combinateurs similaire à Parsec [5] ou Planck [3]. Il y a cependant quelques différences notables, comme l'absence *d'alternative exclusive* ($g_1 < | > \ldots < | > g_n$ en Parsec) qui essaye g_{p+1} uniquement si $g_1,\ldots,g_p$ n'ont consommé aucun caractère. Ce combinateur n'est pas nécessaire pour obtenir de bonnes performances avec `DeCaP`. Notre bibliothèque fournit deux combinateurs d'alternatives qui acceptent tout deux plus d'entrées (voir section 2.3).

2.1. La monade α `grammar`

Les bibliothèques de combinateurs de parseurs disponibles à l'heure actuelle sont basés sur une monade [3] proche de la monade triviale $\alpha\, M = flux \to \alpha \times flux$. Celle de `DeCaP` (listing 3) est quant à elle plus complexe et reprend quatre idées principales :

- La fonction réalisant l'analyse syntaxique utilise un style de passage par continuations.
- On calcule une approximation de l'ensemble des premiers caractères acceptés par une grammaire, ainsi qu'un booléen indiquant si une grammaire accepte l'entrée vide. Ceci permet d'optimiser les combinateurs en éliminant très tôt certaines branches de l'arbre des possibilités.
- On stocke également dans la monade une information utile à la production de messages d'erreurs : la liste des *noms* des terminaux acceptés comme segment initial de la grammaire. La fonction d'analyse maintient donc un enregistrement mutable de type `errInfo` contenant la position maximale atteinte, ainsi que la liste des terminaux qui auraient pu être acceptés à ce point.
- Une fonction de type `blank` éliminant les caractères non significatifs du flux d'entrée est également fournie à la fonction d'analyse. Elle peut être changée facilement en utilisant des combinateurs dédiés, offrant ainsi flexibilité (contrairement à l'analyse lexicale) et simplicité (à l'inverse des bibliothèques de combinateurs habituelles).

Voici les types utilisés par notre monade α `grammar` (les exemples commentés de combinateurs des listings 4 et 6 pourront être très utiles pour comprendre notre monade) :

`flux` est le type d'un flux de caractères. Dans l'implantation courante il s'agit en fait d'un couple `buffer` × `int` où `buffer` est un flux paresseux de chaînes de caractères. L'entier donne la

3. Dans cet article une monade peut être vue comme un type à un paramètre.

Listing 3 – définition de la monade

```
type blank = flux → flux

type errInfo = { mutable maxPos   : int;
                 mutable buf      : flux;
                 mutable messages : string list }

type grouped = { blank   : blank; errInfo : errInfo }

type next = { firstChars : charset; firstSyms  : string list }

type (α, β) continuation = flux → flux → flux → α → β

type α grammar = { acceptEmpty : bool;
                   firstChars  : charset;
                   firstSyms   : string list;
                   parse : β.grouped → flux → next → (α, β) continuation → β }
```

position dans la chaîne au sommet [4].

`blank` est le type des fonctions qui ignorent les caractères non significatifs (blancs, saut de ligne, commentaires, ...). Une telle fonction prend un flux en argument et lui retire éventuellement des caractères non significatifs.

`errInfo` correspond à un enregistrement qui est utilisé pour stocker la position maximale atteinte et le nom des terminaux qui auraient été acceptés après cette position. Un seul enregistrement de ce type est créé quand on analyse un flux.

`grouped` est un type enregistrement qui permet de regrouper diverses informations qui ne sont que très rarement (ou jamais) modifiées. En pratique, la fonction d'élimination des caractères non significatifs change peu souvent, et l'enregistrement `errInfo : errInfo` ne change jamais. L'intérêt ici est de rendre le code plus lisible, en limitant quelque peu le nombre d'arguments transmis à la fonction d'analyse.

`charset` est le type des ensembles de caractères.

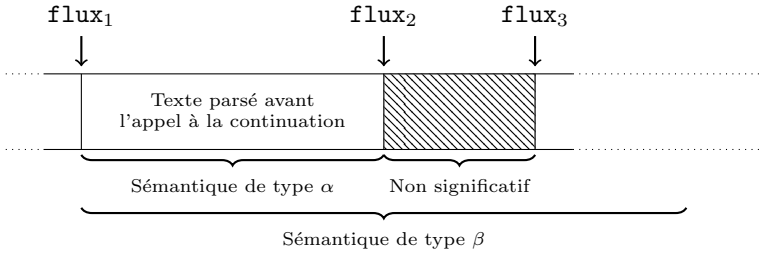
`next` est un enregistrement qui permet de stocker une approximation d'un ensemble de caractères acceptés comme caractère initial d'un langage (l'ensemble peut être trop grand), et le nom des terminaux correspondants (plus de détails sont dans la description de α `grammar`).

(α, β) `continuation` est le type des continuations qui seront appelées après avoir produit une valeur sémantique de type α à partir d'une partie du flux d'entrée. Cette continuation utilise ce résultat tout en continuant l'analyse pour renvoyer un résultat de type β.

La variable β constitue le type de retour de la fonction `parse` décrite ci-dessous, dans laquelle elle est quantifiée universellement. Si on ne regarde que l'aspect *continuation*, on a une monade $\alpha\, M = \alpha \mapsto \forall\beta((\alpha \to \beta) \to \beta)$ qui permet d'encoder les continuations délimitées.

4. Ce type a en fait été curryfié. En réalité, on a (α, β) `continuation` = `buffer` → `int` → `buffer` → `int` → `buffer` → `int` → $\alpha \to \beta$ et `blank` = `buffer` → `int` → `buffer` × `int`.

La continuation reçoit également trois flux en argument. Le premier correspond au flux initial (c'est à dire avant l'analyse retournant une valeur sémantique de type α), le second correspond au flux après l'analyse et le troisième correspond au second auquel on a appliqué la fonction d'élimination des caractères non significatifs.



α `grammar` est le type d'une grammaire renvoyant une valeur de type α lorsqu'elle est employée pour parser le flux d'entrée. Ce type spécifie quatre champs : le booléen `acceptEmpty` indique si la grammaire accepte l'entrée vide, l'ensemble de caractères `firstChars` et la liste `firstSyms` représentent respectivement l'ensemble des caractères et l'ensemble des noms des terminaux acceptés en tête de flux.

La valeur des deux champs `acceptEmpty` et `firstChars` n'est pas toujours exacte. Si `acceptEmpty` $= false$ alors la grammaire n'acceptera jamais la chaîne vide. Le champ `firstChars` doit contenir au moins tous les caractères acceptés, mais il peut en contenir plus [5].

La liste de noms de terminaux `firstSyms` [6] est utilisée uniquement dans les messages d'erreurs, et permet de donner des noms de terminaux à la place d'une liste de caractères attendus au moment de l'échec le plus avancé.

L'argument `next` : `next` de la fonction parse donne les même informations que les champs de même nom dans la structure grammaire, mais se réfère au langage accepté par la continuation. Ceci permet d'optimiser les opérateurs à la Kleene.

2.2. Exemples de combinateurs

Afin d'illustrer le fonctionnement de `DeCaP`, examinons l'implantation de deux combinateurs : `char` et `alternatives`. Le premier est un combinateur atomique, c'est à dire qu'il n'accepte pas de grammaire comme argument. Le second exemple construit quant à lui une grammaire en utilisant une liste de grammaires.

Le listing 4 continent l'implantation du combinateur `char` qui accepte un unique caractère donné en premier argument. En cas de succès, ce combinateur retourne la valeur sémantique donnée en second argument. Dans le cas de ce combinateur trivial, on peut donner une valeur exacte pour les champs `acceptEmpty` et `firstChars`. La fonction `parse` utilise la fonction `read` pour lire un caractère du flux d'entrée. Si il ne correspond pas au caractère attendu, une exception est levée par l'intermédiaire de la fonction `parse_error` (qui sera examinée plus bas). La continuation est ensuite appelée avec quatre arguments : les trois flux attendus ainsi que la valeur sémantique (`a`).

Le listing 5 montre l'implantation de la fonction `parseError` du listing 5, qui est appelée en cas d'échec de l'analyse. Son fonctionnement est relativement simple : la position de l'erreur est extraite du flux d'entrée passé en argument, puis elle est comparée à la position `grouped.errInfo.maxPos`. Si elles sont identiques, le message d'erreur `msg` est ajouté à la liste `grouped.errInfo.messages`. Dans

5. Donner l'ensemble exact requiert de calculer si une grammaire est vide, par exemple, la grammaire {"`a`" g} accepte "`a`" si et seulement si g ne correspond pas au langage vide.

6. Dans l'implantation, un arbre binaire et utilisé pour obtenir une opération d'insertion en temps constant. En cas d'erreur, cet arbre est transformé en liste et les doublons éliminés.

Listing 4 – le combinateur `char`

```
let char : char → α → α grammar = fun s a →
  let name = String.make 1 s in
  { firstChars = singleton s;
    firstSyms = [name];
    acceptEmpty = false;
    parse = fun grouped flux1 next continuation →
              let c, flux2 = read flux1 in
              if c <> s then parseError grouped [name] flux1;
              let flux3 = grouped.blank flux2 in
              continuation flux1 flux2 flux3 a }
```

Listing 5 – gestion des erreurs

```
let recordError grouped msg flux =
  let pos = Input.position flux in
  let pos' = grouped.errInfo.maxPos in
  let c = compare pos pos' in
  if c = 0 then
    grouped.errInfo.messages ← msg @ grouped.errInfo.messages
  else if c > 0 then begin
    grouped.errInfo.maxPos ← pos;
    grouped.errInfo.flux = flux;
    grouped.errInfo.messages ← msg;
  end

let parseError grouped msg flux = recordError grouped msg flux; raise Error
```

le cas où la position de l'erreur courante est plus grande, l'enregistrement `grouped.errInfo` est mis à jour en totalité car une position plus avancée a été atteinte [7].

Intéressons nous maintenant à un combinateur plus complexe : `alternatives` (listing 6). Il permet de réunir plusieurs grammaires au sein d'une seule et même grammaire qui formera leur « réunion ». Les champs `firstChars`, `firstSyms` et `acceptEmpty` sont complétés naturellement : la grammaire produite pourra être vide si et seulement si au moins une de grammaires de la liste accepte l'entrée vide. L'ensemble des premiers caractères et l'ensemble des premiers symboles sont calculés avec de simples unions.

La fonction `parse` est plus intéressante : dans un premier temps, elle fait appel à la fonction `testFirstChars` (donnée plus bas) afin d'éliminer les alternatives qui sont incompatibles avec le premier caractère du flux d'entrée. On vérifie également si la continuation pourrait accepter le flux d'entrée actuel (`emptyOk` = `true`) et, dans ce cas, on conserve aussi les alternatives acceptant l'entrée vide. Ce filtrage est effectué en considérant uniquement les premiers caractères, c'est à dire ceux stockés dans les champs `firstChars` des grammaires, ce qui permet d'éliminer de nombreuses possibilités, en pratique. C'est ce calcul qui rend les alternatives exclusives à la `Parsec` peu utiles pour optimiser l'analyse.

Dans un second temps, la fonction `parse` tente d'analyser le flux d'entrée en utilisant les grammaires restantes une par une, jusqu'à un éventuel succès [8]. Il est important de noter que c'est la fonction `l.parse` qui appellera la continuation. Par conséquent, si une erreur se produit dans la suite de

7. Le flux est conservé dans l'enregistrement `grouped.errInfo` car il contient des informations sur la position (nom de fichier, numéro de ligne...)

8. Ici, le cas [] a été sorti de la fonction récursive `fn`. De ce fait, l'appel à `l.parse` dans le cas où la liste contient un seul élément est terminal. C'est parfois le seul appel si le premier caractère suffit à éliminer toutes les autres alternatives. Ce type de situation apparaît souvent en pratique, et est essentiel pour l'efficacité relative de `DeCaP`.

Listing 6 – le combinateur d'alternatives

```
let alternatives : α grammar list → α grammar = fun ls →
  { firstChars = List.fold_left (fun s p → union s p.fistChars) empty_charset ls);
    firstSyms = List.fold_left (fun s p → s @ p.firstSyms)) Empty ls;
    acceptEmpty = List.exists (fun p → p.acceptEmpty) ls;
    parse = fun grouped flux next continuation →
              let emptyOk = testFirstChars grouped next flux in
              let ls = List.filter (fun g →
                                    (emptyOk && g.acceptEmpty) ||
                                      testFirstChars grouped (nextSyms g) flux) ls in
              if ls = [] then raise Error;
              let rec fn = function
                | [l]   → l.parse grouped str pos next continuation
                | l::ls → (try l.parse grouped flux pos next continuation
                               with Error → fn ls)
              in fn ls }
```

Listing 7 – gestion des erreurs (suite et fin)

```
let nextSyms g = { accepted_char = g.firstChars; first_syms = g.firstSyms; }

let testFirstChars grouped next str =
  let c, _= read str in
  let res = Charset.mem next.firstChars c in
  if not res then begin
    let msg = next.firstSyms in
    recordError grouped msg str
  end; res
```

l'analyse grammaticale, et qu'il reste des alternatives à essayer, elles seront également explorées. Une conséquence de cette propriété est que, par exemple, la grammaires `{'a' | 'a' 'b'}` qui est traduite en :

```
alternatives [ char 'a' (); sequence (char 'a' ()) (char 'b' ()) (fun _ _ → ())]
```

accepte effectivement la chaine `"ab"`, bien que ce ne soit pas le cas avec les combinateurs n'utilisant pas le passage par continuations.

L'implantation du combinateur `alternatives` utilise les deux fonctions définies dans le listing 7. La fonction `nextSyms` sert simplement à construire un enregistrement de type `next` à partir des deux champs du même nom d'un enregistrement de type α `grammar`. La fonction `testFirstChars`, quant à elle, utilise un tel enregistrement pour vérifier qu'un caractère présent dans le flux sera accepté par la suite de l'analyse, c'est à dire par la continuation. Cette fonction doit également enregistrer le nom et la position des symboles rejetés dans `grouped.errInfo` si on veut obtenir un message d'erreur complet dans le cas où la suite de l'analyse ne consomme aucun caractère.

2.3. Langage et sémantique des combinateurs

Les combinateurs de `DeCaP` ont tous une implantation relativement simple, similaire à ceux que l'on vient de détailler. On peut les regrouper de manière naturelle :

Les terminaux ont un type proche de l'*unité* de la monade (`empty` : $\alpha \rightarrow \alpha$ `grammar`), qui parse une portion vide du flux et renvoie $a : \alpha$.

Les séquences qui sont toutes définissables à partir de la *liaison* de la monade (`dependent_sequence` : α `grammar` $\rightarrow (\alpha \rightarrow \beta$ `grammar`$) \rightarrow \beta$ `grammar`).

Les modificateurs de sémantique qui s'appliquent à une seule grammaire et modifient le résultat de l'action sans changer le langage parsé.

Les modificateurs d'agencement qui modifient le langage parsé en jouant sur la définition des caractères non significatifs.

Les constructeurs d'*EBNF* qui viennent en deux versions, délimitée à l'aide du nouveau combinateur `delim`, ou non délimité.

On va maintenant préciser certains combinateurs dans chaque catégorie, en insistant sur ceux qui, à notre connaissance, n'existent pas dans les autres bibliothèques de combinateurs.

Parmi les terminaux, il faut noter la présence de `black_box` qui permet de transformer n'importe qu'elle fonction de type $\texttt{flux} \to \alpha \times \texttt{flux}$ en un terminal de type $\alpha\,\texttt{grammar}$. On doit en plus fournir les informations pour les champs `firstChars`, `firstSyms`, et `acceptEmpty`. Ce combinateur n'est pas vraiment nouveau (`mkPT` dans `Parsec`). Une des utilisations principales de `black_box` est de tester l'entrée, par exemple pour vérifier qu'un mot clé n'est pas suivi d'un caractère alpha-numérique.

Les combinateurs de séquence ne présentent pas d'originalité particulière. La liaison de la monade correspond à $\texttt{dependent\_sequence} : \alpha\,\texttt{grammar} \to (\alpha \to \beta\,\texttt{grammar}) \to \beta\,\texttt{grammar}$. Ce combinateur est moins efficace que la séquence simple $\texttt{sequence} : \alpha\,\texttt{grammar} \to \beta\,\texttt{grammar} \to (\alpha \to \beta \to \gamma) \to \gamma\,\texttt{grammar}$. En effet, si on considère une grammaire $\{g_1\,g_2\}$ traduite en $(\texttt{sequence}\,g_1\,g_2)$ La fonction $g_1.\texttt{parse}$ reçoit les premiers caractères acceptés par g_2. Si g_1 est une alternative, on peut utiliser cette information pour éliminer certains cas. De même, si g_1 est de la forme $\{g_1'*\}$, cette information permet parfois de savoir qu'il est nécessaire d'analyser le flux avec une occurrence supplémentaire de g_1'. Il est bien entendu impossible de coder ce genre d'optimisation avec `dependent_sequence`.

Dans notre syntaxe *EBNF*, $\{a{:}g_1\ b{:}g_2 \twoheadrightarrow (g_3\,a\,b)\}$ dénote une séquence dépendante. Le combinateur qui permet la traduction la plus simple de cette syntaxe est `iter` [9] : $\alpha\,\texttt{grammar}\,\texttt{grammar} \to \alpha\,\texttt{grammar}$ qui est définit de manière immédiate comme $\texttt{iter}\ g = \texttt{dependent\_sequence}\ g\ (\textbf{fun}\,x \to x)$. La traduction de $\{a{:}g_1\ b{:}g_2 \twoheadrightarrow (g_3\,a\,b)\}$ est alors $\texttt{iter}\ (\texttt{sequence}\ g_1\ g_2\ (\textbf{fun}\,a\,b \to g_3\ a\ b))$ qui est facile plus facile à générer par induction.

Les modificateurs de sémantique sont $\texttt{apply} : (\alpha \to \beta) \to \alpha\,\texttt{grammar} \to \beta\,\texttt{grammar}$ et $\texttt{apply\_position} : (\alpha \to \texttt{flux} \to \texttt{flux} \to \beta) \to \alpha\,\texttt{grammar} \to \beta\,\texttt{grammar}$. Ce dernier expose le flux dont on peut extraire la position. La traduction de notre syntaxe *EBNF* l'utilise pour donner accès aux positions automatiquement avec une syntaxe identique à celle de `Camlp4`, mais par contre avec un type libre pour les positions [10].

Listing 8 – Exemple de modification d'agencement

```
let inter_paragraph = Decap.blank_regexp  ''[ \t\r\n]*''
let inter_word str pos =
  let gram = parser ''[ \t\r]*'' ''\n''? ''[ \t\r]*'' in
  let str, pos, _ = Decap.partial_parse_buffer gram Decap.no_blank str pos in
  str, pos

let word = parser w:''[^ \n\t\r]+'' → w
let paragraph = Decap.change_layout (parser ws:word+ → ws) inter_word
let text = parser ps:paragraph* → ps

let _ =
  let ps = Decap.handle_exception (Decap.parse_channel text inter_paragraph) stdin in
  Printf.printf "%i␣paragraphs␣read.\n" (List.length ps)
```

9. Le combinateur `iter` correspond au *join* des monades. On dispose aussi de `apply` qui correspond à *fmap*.
10. On doit spécifier une fonction de type $\texttt{flux} \to \texttt{flux} \to \texttt{position}$ en utilisant `#define LOCATE ma_fonction`.

Il y a aussi deux modificateurs d'agencement : `ignore_next_blank` : α `grammar` $\rightarrow$ α `grammar` supprime l'utilisation de la fonction `blank` après une grammaire. Cela permet d'interdire les espaces entre deux grammaires d'une séquence. Ainsi `sequence` (`ignore_next_blank` g_1) g_2 interdira les espacements entre g_1 et g_2. D'autre part, `change_layout` : α `grammar` $\rightarrow$ `blank` $\rightarrow$ α `grammar` change les caractères non significatifs d'une grammaire.

En général, la gestion des caractères non-significatifs introduit une certaine lourdeur lorsque l'on utilise des grammaires avec des combinateurs. Si l'on analyse ces caractères après tous les terminaux de la grammaire (et au début du flux), cela n'influe que sur la définition de chaque terminal. Avec `DeCaP`, tous les combinateurs reçoivent et retransmettent la fonction `blank`. On doit seulement la fournir une fois au début de l'analyse, et lorsqu'on en change.

Par ailleurs, `DeCaP` fournit une fonction `partial_parse_buffer` de type α, `grammar` $\rightarrow$ `blank` $\rightarrow$ `flux` $\rightarrow$ `flux` $\times$ α que l'on peut facilement utiliser pour définir une fonction `blank` : `flux` $\rightarrow$ `flux` à partir d'une grammaire (voir définition de `inter_word`[11], listing 8).

Le fait de pouvoir changer la fonction `blank` permet d'écrire de manière naturelle une grammaire pour analyser un texte constitué d'une séquence de paragraphes séparés par au moins une ligne vide. Il suffit de changer la fonction `blank` des paragraphes pour qu'elle accepte au plus un saut de ligne (voir listing 8).

En fait, la grammaire du listing 8 fait déborder la pile sur les textes très long. Cela est du au fait que la grammaire des paragraphes accepte aussi les segments initiaux de chaque paragraphe. Pour éviter cela, on pourrait analyser explicitement les espaces entre paragraphes, mais si la grammaire s'enrichit, par exemple avec des sections, ce n'est plus très naturel. Une meilleure solution est d'empêcher la grammaire des paragraphes de revenir en arrière après avoir accepté un mot. Pour cela il suffit d'utiliser le combinateur `delim` : α `grammar` $\rightarrow$ α `grammar` qui interdit de revenir en arrière sur le résultat d'une grammaire dès qu'elle a analysé complètement une portion du flux.

Derrière les abréviations *EBNF* `g*` et `g+` se trouve les combinateurs `fixpoint` et `fixpoint1` de type $\alpha \rightarrow (\alpha \rightarrow \alpha)$ `grammar` $\rightarrow$ α `grammar`. Par défaut, `{'a'* 'a' 'a'}` accepte les chaînes avec au moins deux « `a` »[12], mais `'a'*` doit alors revenir en arrière deux fois.

Cette possibilité de retour en arrière, qui est nécessaire pour obtenir la sémantique des opérateurs à la Kleene, pose parfois des problèmes d'efficacité. En utilisant `delim` on peut écrire deux combinateurs `fixpoint'` a g = `fixpoint` a (`delim` g) et `fixpoint1'` a g = `fixpoint1` a (`delim` g) qui vont analyser des séquences de g, sans aucun retour en arrière possible une fois que g a réussi à analyser une portion du flux. Une remarque similaire peut être faite pour l'opérateur d'alternatives `|`. Le combinateur ainsi obtenu est toujours plus général que l'opérateur d'alternatives exclusives `<|>` de parsec, que nous n'avons pas eu besoin de définir.

L'implantation de `delim` est particulièrement simple :

Listing 9 – Le combinateur `delim`

```
let delim : 'a grammar → 'a grammar = fun l →
  { firstChars = lazy (firstChars l);
    firstSyms = lazy (firstSyms l);
    acceptEmpty = lazy (Empty l);
    parse = fun grouped flux next cont →
              let cont' flux flux' flux'' x () = cont flux flux' flux'' x in
              l.parse grouped flux next cont' () }
```

11. On ne peut pas écrire simplement la fonction `inter_word` avec `blank_regexp ''[ \t\r]*[\n]?[ \t\r]*''` car cette fonction applique l'expression régulière à chaque ligne du flux jusqu'à un échec. Ceci est une conséquence de la représentation des flux (listes paresseuses de lignes) et du fait que le module `Str` ne traite que les chaînes. Ainsi, les terminaux de `DeCaP` utilisant des expressions régulières ne peuvent pas analyser les sauts de ligne, qu'il faut malheureusement lire comme des caractères ou des chaînes.

12. Bien sur, cette grammaire doit plutôt être écrite `{'a' 'a' 'a'*}`.

Il suffit que la continuation ait un argument de plus de type `unit` pour qu'elle ne soit effectivement appelée qu'après que `l.parse` ait retourné un résultat. Ainsi, on empêche `l.parse` de revenir en arrière en cas d'erreur dans la continuation. Le combinateur `delim` correspond très exactement à la fonction `reset` des continuations délimitées.

3. Langages récursifs

En général, les combinateurs de parseur permettent de traiter les grammaires récursives comme de simples fonctions récursives. Il y a cependant une limitation : la récursion à gauche est impossible car elle consiste en une fonction qui boucle en s'appelant elle même sans consommer le flux d'entrée.

3.1. Récursion à droite

La version donnée de la monade ne permet pas de calculer l'ensemble des premiers caractères acceptés par une grammaire récursive à droite, ni de savoir si une telle grammaire accepte l'entrée vide. Il y a deux raisons :

— Le langage `OCaml` limite la définition des structures récursives, on se heurte souvent au message « This kind of expression is not allowed as right-hand side of 'let rec' ». Une manière de résoudre ce problème est de rendre mutable les champs de la structure α `grammar` et de fournir deux fonctions `declare_grammar` : `string` $\to$ α `grammar` et `set_grammar` : α `grammar` $\to$ α `grammar` $\to$ `unit` pour la déclaration et la définition des grammaires récursives. Cela permet également de découper la définition d'une grammaire récursive sur plusieurs fichiers.

— La fonction `declare_grammar` n'a aucune information sur la future grammaire, on doit donc rendre les trois champs `acceptEmpty`, `firstChars` et `firstSyms` de la structure α `grammar` paresseux pour ne les évaluer qu'après avoir défini la grammaire.

Les changements à effectuer pour autoriser la définition de grammaires récursives sont résumés dans Le listing 10.

Listing 10 – Modifications de la monade pour la récursion

```
type α grammar = { acceptEmpty : bool lazy;
                   firstChars : charset lazy;
                   firstSyms : string list lazy;
                   parse : β.grouped → flux → next → (α, β) continuation → β }

let not_ready name _ = failwith ("not_ready:␣"^name)

let declare_grammar name = { acceptEmpty = lazy (not_ready name ());
                             firstChars = lazy (not_ready name ());
                             firstSyms = lazy (not_ready name ());
                             parse = (not_ready name); }

let set_grammar p1 p2 =
  p1.firstChars <- p2.firstChars;
  p1.acceptEmpty <- p2.acceptEmpty;
  p1.firstSyms <- p2.firstSyms;
  p1.parse <- p2.parse
```

Des fonctions pour les grammaires récursives paramétrées sont également fournies. Elles mémorisent la structure α `grammar` dans une table indexée par les valeurs des paramètres. Cette opération permet de ne pas reconstruire la structure à chaque fois que la grammaire est utilisée, ce qui dégraderait grandement les performances. Cette remarque est également valable pour les grammaires paramétrées non récursive, on a donc souvent intérêt à mémoriser toutes les grammaires paramétrées.

Cette nouvelle définition de la monade ne fonctionne que lorsqu'on se restreint aux grammaires qui ne sont pas récursives à gauche. En effet, calculer la valeur des champs `acceptEmpty` et `firstChars` revient en général à calculer un point fixe sur la définition récursive de la grammaire. Étant donné que ces informations dépendent uniquement du premier caractère accepté par la grammaire, le calcul du point fixe se fait en une seule étape en l'absence de récursion à gauche. Dans ce cas, le type paresseux est suffisant.

3.2. Récursion à gauche

Nous avons aussi exploré des solutions pour lever la limitation de la récursion à gauche. Considérons la définition récursive $g = F\,g$ d'une grammaire récursive g comme point fixe d'une grammaire F paramétrée par une autre grammaire. On peut alors donner en paramètre à F une grammaire s qui collecte la continuation passé à s.`parse` dans une liste L_0 avant d'échouer. On analyse alors la chaîne vide avec la grammaire $F\,s$ afin d'obtenir une liste de continuations L_0 que l'on peut aisément transformer en une liste de grammaires L. Le fait que s échoue garantit que l'on explore la grammaire F complètement (si $F\,s$ accepte la chaîne vide uniquement en dernière possibilité). Si on suppose, pour simplifier, que toutes les occurrences de g dans $F\,g$ sont récursives à gauche, g est équivalentes à $\{(F$ `FAIL`$)$ (`alternatives` $L)*\}$. Dans le cas général, seules les occurrences récursives à gauche de g dans $F\ g$ doivent être remplacées par `FAIL`.

Cette approche semble fonctionner, mais elle pose plusieurs problèmes importants :

— Elle n'est pas typable avec la monade actuelle. Lorsque l'on collecte les continuations avec la grammaire s, la fonction `parse` de s doit stocker dans L_0 son argument `continuation` : $\alpha \to \beta$ pour un type β donné qui peut varier. Il faudrait en fait remplacer β par le type vide.

— Si l'on remplace β par le type vide, le combinateur `delim` n'est plus typable. Passer outre le typage est ici incorrect. La solution proposée est donc incompatible avec la notion de délimitation, pourtant essentielle à l'optimisation des grammaires.

— Le calcul de point fixe nécessaire pour calculer les premiers caractères acceptés par la grammaire est plus complexe. Dans notre test, on a conservé, pour chaque grammaire, la liste des grammaires qui la référencent. On peut ainsi calculer le point fixe par des parcours de graphe. L'initialisation de la grammaire est significativement plus coûteuse, ce qui peut être gênant lorsque l'on analyse de multiples fichiers de petite taille, ou lorsque l'on a des grammaires paramétrées qui doivent être construite pour un grand nombre de valeurs du paramètre.

Nous avons donc renoncé à cette approche de récursion à gauche pour l'instant trop compliquée, et inefficace du fait de l'incompatibilité avec les délimitations.

4. Évaluation des performances

Nous avons testé notre approche sur des grammaires simples ainsi que sur la grammaire d'`OCaml` et son extension pour `Patoline`. Commençons par examiner la calculatrice donnée au début de l'article (Listing 1). Bien que très simple, cette grammaire est intéressante du fait que notre prédicteur ne suffit pas pour distinguer le produit la puissance qui commence tout deux par une étoile.

Pour obtenir des éléments de comparaison, nous avons traduit cette grammaire vers `OCamlYacc` et `Camlp4`. Nous avons ensuite comfronté ces dernières à deux versions de `DeCaP` : la version donnée dans le listing 1 (nommée « bnf » dans la figure 1) et une variante (nommée « gourmande ») qui utilise l'opérateur délimité « $**$ » à la place de « $*$ » uniquement pour les produits.

On constate que tous les programmes fonctionnent en temps et mémoire linéaire, sauf la grammaire `DeCaP` non modifiée. Cette grammaire n'effectue pourtant aucun retour en arrière, mais elle conserve une quantité linéaire de données en mémoire, qui pourraient être utilisées pour un éventuel retour en arrière. En effet, sur la base du premier caractère, on ne peut pas éliminer l'éventualité de devoir

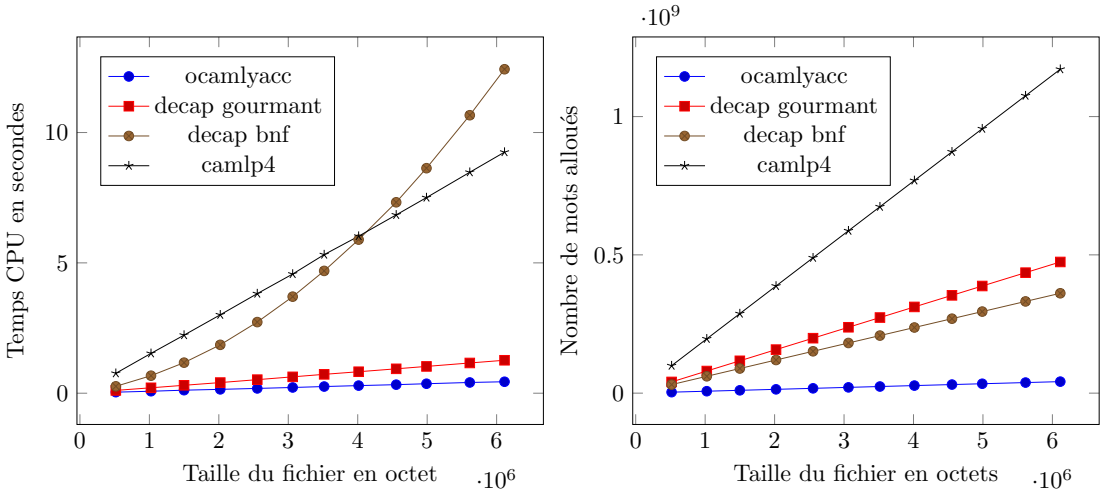


FIGURE 1 – Performances de la calculatrice

revenir en arrière (du fait de la présence des deux opérateurs « * » et « ** »).

Ainsi, une clôture contenant un pointeur sur le flux est conservée pour chaque produit. Lors d'une exécution, on voit très bien que la quantité de mémoire allouée au processus ne cesse de grimper pour cette grammaire, ce qui n'est pas le cas pour toutes les autres. Sur un fichier de 6Mo le processus consomme plus de 400Mo, alors que la grammaire utilisant une délimitation n'utilise jamais plus de 20Mo.

Avoir une quantité de mémoire accessible croissant linéairement au cours du temps rends la plupart des ramasses miettes (celui d'OCaml inclus) quadratique en temps. Il s'agit de la seule source de non linéarité pour cette grammaire. Ceci est un problème récurrent pour les parseurs à base de combinateurs utilisés sur de très grands fichiers.

On remarque qu'une fois ce problème corrigé en empêchant le retour en arrière pour les produits, la grammaire `DeCaP` se comporte plutôt bien en temps (environ 3 fois plus lent que la version `OCamlYacc` et 7 fois plus rapide que `Camlp4`). Toutefois, au niveau du nombre d'allocations, la grammaire `OCamlYacc` se détache nettement, ce qui n'est pas étonnant du fait qu'un automate à pile alloue beaucoup moins de mémoire que le code fonctionnel d'une bibliothèque de combinateurs, qui crée entre autre beaucoup de clôtures.

Le second exemple (figure 2) analyse les performances de notre parseur sur la grammaire complète d'`OCaml`. Cette dernière emploie uniquement des combinateurs non délimités, sauf dans trois cas : les structures, les signatures et les listes de cas du filtrage. Ces trois optimisations suffisent et nos expériences ont montré que délimiter plus de combinateurs ne change pas les performances. Utiliser au maximum les vrais combinateurs à la Kleene (c'est à dire non délimités) simplifie l'écriture de la grammaire car l'ordre des règles n'est pas significatif.

Toutefois, la grammaire d'`OCaml` fonctionne également avec la sémantique gourmande pour tous les combinateurs, à l'exception d'un cas : pour analyser un constructeur d'un module, on a besoin de revenir en arrière. L'expression `A.B.C` est d'abord analysé comme le module `A.B.C` avant que l'absence de point après `C` provoque une erreur et conduise à l'analyser comme le constructeur `C` du module `A.B`. Ce retour en arrière nuit peu aux performances, mais donne dans ce cas précis une règle plus naturelle que dans la grammaire `OCamlYacc` d'origine.

La figure 2 montre les performances de notre grammaire, de celle d'origine et de celle de `Camlp4` sur plusieurs dizaines de fichiers. On montre aussi les ratios de vitesse minimum, moyenne et maximum (les ratios minimums et maximums ont été calculés sur les fichiers d'au moins 8Ko). On constate que les performances obtenues sont intermédiaires entre `Camlp4` et `OCamlYacc`. Elles sont donc suffisantes étant donné que, même pour `Camlp4`, le temps d'analyse syntaxique est petit devant le temps de compilation (très petit quand on compile vers du code natif).

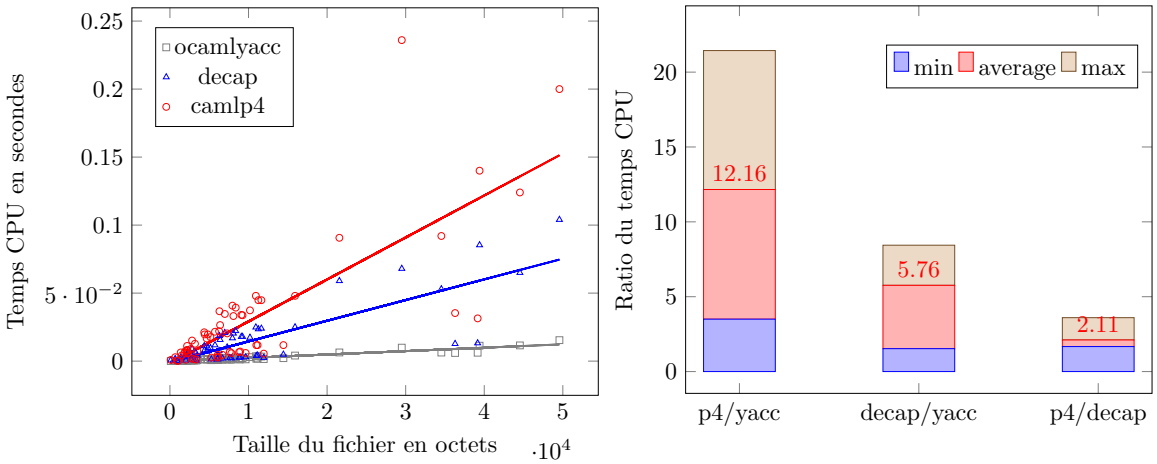


FIGURE 2 – Performances de la grammaire d'OCaml

5. Conclusion et travaux futurs

Nous pensons que `DeCaP` est déjà un bon outil pour écrire de nombreux langage et une alternative intéressante à `Camlp4`. Il reste juste à completer les quotations et anti-quotations, ainsi que de finir d'intégrer les extensions de la version 4.02 d'`OCaml`.

Dans le cas des grands fichiers, pour être une alternative viable aux parseurs générant des automates à pile tels qu'`OCamlYacc`, il est encore nécessaire de gagner en efficacité. Une possibilité serait de remplacer les prédictions sur un seul caractère par un automate fini, afin de se rapprocher des performances de menhir sur une grammaire $LR(1)$ sans ambigüité. Il ne resterait alors probablement plus que le ramasse miette comme sur-coût.

Nous envisageons aussi de tester `DeCaP` sur une grammaire pour une langue naturelle en reprenant un travail précédant réalisé avec `Dypgen` qui avait été interrompu pour des problèmes de performance.

Références

[1] D. Boulytchev. Ostap : Parser combinator library and syntax extension for objective caml. 2009.

[2] D. De Rauglaudre. Camlp4 - reference manual. Technical report, 2003.

[3] J. Furuse. Planck : A small parser combinator library for ocaml. Technical report, sizehttp://camlspotter.blogspot.fr/2011/05/planck-small-parser-combinator-library.html, 2011.

[4] P. Koopman and R. Plasmeijer. Efficient combinator parsers. In *Implementation of Functional Languages*, volume 1595 of *LNCS*, pages 120–136. 1999.

[5] D. Leijen and E. Meijer. Parsec : Direct style monadic parser combinators for the real world. Technical report, Department of Computer Science, Universiteit Utrecht, 2001.

[6] R. Lepigre and C. Raffalli. Getting started writing parsers and syntax extensions using decap pa_ocaml. Technical report, sizehttp://lama.univ-savoie.fr/decap/, 2014.

[7] S. Marlow and A. Gill. Happy User Guide. Technical report, sizehttp://www.haskell.org/happy/, 2001.

[8] P.-E. Meunier, C. Raffalli, R. Lepigre, T. Hirschowitz, F. Hatat, P. Hyvernat, and G. Theyssier. Patoline : a modern digital typesetting system, 2011.

[9] E. Onzon. Dypgen : Self-extensible parsers and lexers for ocaml, 2005.

[10] M. Pellauer, M. Forsberg, and A. Ranta. BNF Converter : Multilingual Front-End Generation from Labelled BNF. Technical report, Department of Computing Science, Chalmers University of Technology and Göteborg University, 2004.

[11] F. Pottier and Y. Régis-Gianas. Menhir. Technical report, sizehttp://gallium.inria.fr/~fpottier/menhir/, 2005.

[12] S. D. Swierstra and P. R. Azero Alcocer. Fast, error correcting parser combinators : a short tutorial. In J. Pavelka, G. Tel, and M. Bartosek, editors, *SOFSEM'99 Theory and Practice of Informatics, 26th Seminar on Current Trends in Theory and Practice of Informatics*, volume 1725 of *LNCS*, pages 111–129, November 1999.