

Un aperçu de Menhir

François Pottier et Yann Régis-Gianas

30 mars 2006



Motivation

L'avant

Le milieu

L'arrière

Conclusion

Motivation

Pourquoi un nouveau générateur d'analyseurs syntaxiques?

- ▶ d'abord pour tester expérimentalement nos idées sur le *typage* des parseurs LR;
- ▶ ensuite parce que l'envie ne manquait pas d'améliorer *ocamlyacc*.

Motivation

L'avant

Le milieu

L'arrière

Conclusion

Traits marquants

Le langage de spécification offre de nouvelles possibilités, parmi lesquelles:

- ▶ définir des *non-terminaux paramétrés*;
- ▶ découper une spécification en plusieurs « *modules* »;
- ▶ demander le développement « *en ligne* » d'un non-terminal;

Ces traits permettent l'écriture d'une *bibliothèque standard*.

Définition d'un non-terminal paramétré

Voici une définition typique, issue de la librairie standard:

```
option(X):  
  | { None }  
  | x = X { Some x }
```

On remarque au passage que les valeurs sémantiques sont *nommées*.

Emploi d'un non-terminal paramétré

Voici comment on pourrait définir une liste de déclarations, chacune suivie d'une virgule *optionnelle*:

declarations:

```
| { [] }  
| d = declaration; option(COMMA);  
  ds = declarations { d :: ds }
```

On peut écrire *COMMA?* au lieu de *option*(*COMMA*).

Expansion des non-terminaux paramétrés

Tout se passe comme si on avait écrit:

option_comma:

| { None }
| x = COMMA { Some x }

declarations:

| { [] }
| d = declaration; *option_comma*;
| ds = declarations { d :: ds }

Toute référence à un non-terminal paramétré est *expansée*.

Autres exemples

La librairie standard définit également:

terminated(X, closing):

| $x = X; \text{closing} \{ x \}$

list(X):

| $\{ [] \}$

| $x = X; xs = \text{list}(X) \{ x :: xs \}$

Notre *declarations* précédent peut alors s'écrire

list(terminated(declaration, COMMA ?))

Est-ce tout-à-fait équivalent?

Expansion en ligne

Les perfectionnistes pourront déclarer:

```
%inline terminated(X, closing):  
  | x = X; closing { x }
```

La définition de *terminated* sera alors expansée « en ligne » et une réduction inutile sera évitée.

Un exemple plus convaincant

L'expansion en ligne permet *d'éviter certains conflits* LR(1) sans abîmer la grammaire. Exemple tiré de la grammaire de Menhir:

producer:

```
| id = ioption(terminated(LID, EQUAL));  
  p = actual_parameter  
  { id, p }
```

ioption est identique à *option* mais est déclaré **%inline**. Le terminal *LID* appartient à `FIRST(actual_parameter)`.

Modularité

Menhir propose une forme faible de modularité:

- ▶ chaque fichier forme un « *(mixin) module* »;
- ▶ une grammaire complète est obtenue par *composition*;
- ▶ les non-terminaux déclarés **%public** sont *partagés*, les autres sont considérés comme *internes*;
- ▶ la définition d'un non-terminal public peut être *répartie* sur plusieurs modules.

La librairie standard est un module comme un autre.

Traits de moindre intérêt

Menhir permet également:

- ▶ d'utiliser un type token *externe*;
- ▶ de *partager* un type token entre plusieurs grammaires;
- ▶ de *paramétrer* l'analyseur par un *module* Objective Caml;
- ▶ de *typer* les actions sémantiques avant la génération.

Motivation

L'avant

Le milieu

L'arrière

Conclusion

Techniques de construction

Voici un petit rappel:

- ▶ les items $LR(0)$ ont la forme $A \rightarrow a \bullet \beta$; il y a conflit $LR(0)$ dès qu'un décalage et une réduction, ou bien deux réductions, sont permis dans le même état;
- ▶ les items $LR(1)$ ont la forme $A \rightarrow a \bullet \beta [T]$ où T est l'ensemble des terminaux susceptibles de suivre.

L'automate *non-déterministe* $LR(1)$ est de taille linéaire... mais la détermination explose en pratique.

Techniques de construction

Pour approcher l'expressivité de LR(1) sans en payer le prix:

- ▶ la technique SLR(1) consiste à conserver l'automate LR(0), mais à attribuer à chaque item $A \rightarrow \dots$ l'ensemble de prédiction $FOLLOW(A)$;
- ▶ la technique LALR(1) consiste conceptuellement à construire l'automate LR(1) puis à *identifier* les états qui ont le même noyau LR(0);
- ▶ la technique de Pager consiste à construire effectivement l'automate LR(1), mais en *identifiant* au vol les états de même noyau *si on peut garantir que cela ne fera apparaître aucun conflit.*

Techniques de construction

Menhir implante la technique de Pager. En pratique, *l'automate obtenu est l'automate LALR(1)* si la grammaire est LALR(1) et a plus d'états sinon — mais ce n'est pas un théorème.

Implantation un peu subtile, mais après optimisation le bottleneck n'est plus là.

Explication des conflits

J'ai implanté un algorithme *d'explication de conflits* fortement inspiré de celui de DeRemer et Pennello — mais adapté pour Pager au lieu de LALR(1).

Une grammaire à conflits

%token IF THEN ELSE

%start < expression > expression

%%

expression:

```
| ...  
| IF b = expression THEN e = expression { ... }  
| IF b = expression  
|   THEN e = expression  
|   ELSE f = expression { ... }  
| ...
```

Explication des conflits

L'outil rapporte que le conflit est atteint après avoir lu la phrase (*sentential form*) suivante:

IF expression THEN IF expression THEN expression

et lorsque le terminal suivant est *ELSE*.

Explication des conflits

Si on veut savoir plus, on peut demander pourquoi il est légal de *décaler*:

expression

IF expression THEN expression

IF expression THEN expression . ELSE expression

et pourquoi il est légal de *réduire*:

expression

IF expression THEN expression ELSE expression // lookahead token appears

IF expression THEN expression .

Explication des conflits

Évidemment, la *présentation textuelle* de ces arbres n'est pas tout-à-fait au point (on dérive trop vers la droite).

Pour adoucir le problème, Menhir factorise le *contexte maximal* commun aux deux (ou plus) dérivations.

On doit encore pouvoir faire un peu mieux — à suivre.

Motivation

L'avant

Le milieu

L'arrière

Conclusion

Réentrance

Les analyseurs produits par Menhir sont *réentrants* — ils ne reposent sur aucune variable globale.

Cela impose d'abandonner l'API du module Parsing actuel.

L'accès aux positions se fait maintenant par mots-clef.

Un fragment de code engendré

Menhir engendre du *code* (commenté!), non des tables. Extrait:

```
let goto_reversed_list_rule env stack v =  
  let stack = (stack, v) in  
  let tok = env.token in  
  match tok with  
  | EOF →  
    (* Shifting (EOF) to state 17 *)  
    (* Not allocating top stack cell *)  
    (* Reducing without looking ahead at # *)  
    (* Reducing production trailer → EOF *)  
    let v : ( Syntax.trailer option ) = ( None ) in  
    goto_trailer env stack v  
  | ...
```

Code ou tables?

Nous produisons du code parce que le *typage* fin était notre motivation initiale. Toutefois, le code actuel contient encore des *magic* – on attend les GADTs!

Le code produit est optimisé pour la taille, mais reste assez gros.

Un second back-end produisant des tables est concevable.

Quelques chiffres

La grammaire d'Objective Caml: 1.5 Kloc. Automate LALR(1) à 1040 états.

Avec *ocamlyacc*, le *.ml* engendré fait 8 Kloc (400 Kb), le *.o* engendré fait **215 Kb**.

Avec Menhir, le *.ml* engendré fait 30 Kloc (1.5 Mb; plus de 1100 fonctions récursives), le *.o* engendré fait **865 Kb**.

Chiffres obtenus avec *-noassert* et *-compact*. Le code produit par Menhir serait plus gros si les positions étaient correctement traitées.

Gestion d'erreurs

Menhir implante un mécanisme compatible en apparence avec celui d'ocaml yacc (pseudo-terminal *error*). L'implantation est un peu modifiée:

- ▶ la réduction sur *error* est effectuée;
- ▶ la resynchronisation est optionnelle.

Faut-il laisser tomber ce mécanisme (qui me semble *quasi impossible à utiliser correctement*) et adopter une autre approche?

Gestion d'erreurs: suggestions

On pourrait laisser tomber la reprise programmable sur erreur et se contenter de *construire un message* dépendant de l'état atteint.

Nous songeons à:

- ▶ construire automatiquement un message basé sur les *items LR(1)* qui forment l'état atteint;
- ▶ permettre au programmeur d'associer des messages personnalisés aux états à l'aide *d'exemples* (cf. C. Jeffery).

Motivation

L'avant

Le milieu

L'arrière

Conclusion

Feedback!