Analyse d'entrées incorrectes avec Menhir

Frédéric Bour

7 décembre 2015

Plan

1. Motivations
2. Menhir
3. Complétion de l'AST
   - Dérivation d'un symbole
   - Complétion d'un parseur
4. Reprise de l'analyse

Analyser pendant l'édition

Rendre le développement interactif :

- pour assister le développeur,
- raccourcir le cycle édition-compilation-correction,

En parallèle, une tendance : le «tooling» aussi important que le langage (e.g. Go).

Exemples

Pour OCaml, Merlin

- explorer le projet et les bibliothèques,
- complétion, report interactif des types.

Le texte est un medium, mais l'interaction est plus structurée.

Exemples

Pour OCaml, Merlin

- explorer le projet et les bibliothèques,
- complétion, report interactif des types.

Le texte est un medium, mais l'interaction est plus structurée.

Assistants de preuve

- Considérations similaires, mais contexte bien plus riche.
- Impossible à suivre en mode *batch*.

Dans Coq, la granularité est la commande.

En pratique

Quand on lance le *frontend* au milieu de l'édition :

En pratique

Quand on lance le *frontend* au milieu de l'édition :

Lexing OK, erreurs très locales.

En pratique

Quand on lance le *frontend* au milieu de l'édition :

- **Lexing** OK, erreurs très locales.
- **Parsing** Abandon, pas d'AST.

En pratique

Quand on lance le *frontend* au milieu de l'édition :

- **Lexing** OK, erreurs très locales.
- **Parsing** Abandon, pas d'AST.
- **Typing** Acceptable, on ne perd pas d'informations.

Développements en cours

Outils génériques :

- pour le parsing, extension de Menhir
- pour l'interaction, bibliothèques réutilisables pour s'intégrer aux éditeurs.

Si possible, avec un minimum d'intervention de la part du développeur du langage.

Un parser résilient

Ce qu'on attend de ce parser :

completion au minimum, conserver ce qui a déjà été analysé

reprise si possible, continuer d'analyser le reste de l'entrée

Outillage actuel : Yacc

En cas d'erreur, jeter le contenu de la pile jusqu'à trouver un état acceptant le terminal spécial `error`.

- du contenu est perdu !
- il faut étendre la grammaire avec `error`, risque de conflits.
- pas de garantie que cela se passe mieux après cette transition.

Plan

1 Motivations

2 Menhir

3 Complétion de l'AST
- Dérivation d'un symbole
- Complétion d'un parseur

4 Reprise de l'analyse

Menhir, la grammaire

Dans les grammaires : des attributs, similaires à ceux d'OCaml.

```
expr [@recovery default_expr]:
  simple_expr %prec below_SHARP
     $1
| simple_expr simple_labeled_expr_list
     mkexp $startpos $endpos (Pexp_apply($1, List.rev $2))
...
```

Menhir, le générateur

Menhir les ignore, mais génère un `CMLY`.
C'est une représentation synthétique :

- de la grammaire,
- des attributs,
- des automates LR(0) et LR(1).

Menhir, le SDK

Une bibliothèque, similaire aux compiler-libs, permet de traiter ces `CMLY` :

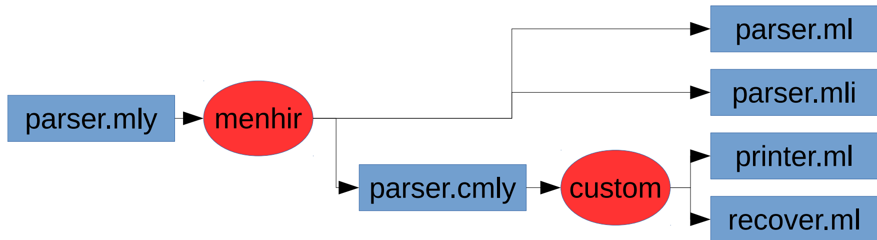
- meta-programmation sur les grammaires (par exemple, implémenter des «printers» génériques)
- l'algorithme de reprise sur erreur est implémenté ainsi.

Menhir, l'interprète

Pour satisfaire aux besoins de l'algorithme de reprise, l'interprète est étendu avec quelques primitives :

- suivre une transition sur un non-terminal,
- forcer une réduction spécifique.

Chaîne de traitement



Plan

1 Motivations

2 Menhir

3 Complétion de l’AST
- Dérivation d’un symbole
- Complétion d’un parseur

4 Reprise de l’analyse

Dérivation d'un symbole

Objectif : pour chaque symbole, produire une séquence d'actions (`SHIFT`, `REDUCE`) qui a pour effet de le pousser sur le haut de la pile.

Dérivation d'un symbole

Objectif : pour chaque symbole, produire une séquence d'actions (`SHIFT`, `REDUCE`) qui a pour effet de le pousser sur le haut de la pile.

```
| pattern WHEN . seq_expr MINUSGREATER seq_expr
```

Dérivation d'un symbole

Objectif : pour chaque symbole, produire une séquence d'actions (SHIFT, REDUCE) qui a pour effet de le pousser sur le haut de la pile.

```
| pattern WHEN . seq_expr MINUSGREATER seq_expr
```

```
derive(seq_expr) = SHIFT, SHIFT, ..., REDUCE
```

Dérivation d'un symbole

Objectif : pour chaque symbole, produire une séquence d'actions (SHIFT, REDUCE) qui a pour effet de le pousser sur le haut de la pile.

```
| pattern WHEN . seq_expr MINUSGREATER seq_expr

derive(seq_expr) = SHIFT, SHIFT, ..., REDUCE

| pattern WHEN seq_expr . MINUSGREATER seq_expr
```

Dérivation d'un symbole

Terminaux

non-paramétré immédiat, `SHIFT` «classique»

paramétré tributaire d'un attribut `recovery`

Dérivation d'un symbole

Terminaux

non-paramétré immédiat, `SHIFT` «classique»

paramétré tributaire d'un attribut `recovery`

Non-terminaux

avec `recovery` immédiat, `SHIFT` «étendu»

sans `recovery` ...

Dérivation d'un non-terminal

Le travail se fait sur la grammaire :

- on part de toutes les productions se réduisant vers ce non-terminal
- on cherche récursivement à dériver chacun des producteurs
- si l'un des symboles est déjà en train d'être dérivé sur la branche actuelle, abandon (produire `expr` nécessite de produire `expr`, nous faisons fausse route).

Coût d'une dérivation

Pour choisir une dérivation :

- un coût est calculé (pour les feuilles, l'auteur de la grammaire peut fournir une annotation `cost`).
- la dérivation de coût minimal est retenue. Un symbole impossible à dériver a un coût infini.

Un outil est fourni pour observer les dérivations choisies et rapporter les dérivations impossibles (manque d'annotations).

Réduction depuis un état

Le travail se fait maintenant sur l'automate.
Partant de l'état courant, il faut trouver une séquence d'actions qui provoque une réduction.
Pour chaque état, on choisit un item de son itemset que l'on va chercher à réduire.
Le choix de l'item est important pour ne pas boucler !

Terminaison

Chaque réduction pousse un symbole sur la pile. Il suffit de choisir les items qui vont consommer au moins deux symboles pour garantir de terminer !

Terminaison

Chaque réduction pousse un symbole sur la pile. Il suffit de choisir les items qui vont consommer au moins deux symboles pour garantir de terminer !

Et si tous les items sont en position 1 ?

```
expr ::= expr . INFIXOP expr
```

Réduire `expr` produit `expr`, on stagne.

Établir un ordre sur les successeurs

- Réduire un item en position 1 amènera l'automate à suivre une transition de son prédécesseur.
- On va ordonner tous ses successeurs, en fonction du nombre de transitions à suivre pour aboutir à un item de position supérieure à 1.
- Il en existe toujours un, sinon le langage reconnu serait vide.

Coût de la complétion

Comme pour la dérivation :

- un coût est calculé pour chaque item.
- la complétion de coût minimal est retenue. Un état impossible à réduire a un coût infini.

Cette fois, un coût infini est une erreur rapportée à l’utilisateur.

Cas particuliers

Imiter Yacc

```
let_bindings ::= let_bindings AND . let_binding
```

On ajoute l'action `POP`.

Cas particuliers

Imiter Yacc

```
let_bindings ::= let_bindings AND . let_binding
```

On ajoute l'action `POP`.

Attribut explicite ou dérivation ?

```
fun_def ::= pattern . fun_def
fun_def ::= -> seq_expr
```

Une `fun_def` est une expression, mais la dérivation introduit des étapes intermédiaires.

Cas particuliers

Imiter Yacc

```
let_bindings ::= let_bindings AND . let_binding
```

On ajoute l'action `POP`.

Attribut explicite ou dérivation ?

```
fun_def ::= pattern . fun_def
fun_def ::= -> seq_expr
```

Une `fun_def` est une expression, mais la dérivation introduit des étapes intermédiaires.

Les actions sémantiques lançant une exception doivent être annotées d'un coût infini !

Plan

1 Motivations

2 Menhir

3 Complétion de l'AST
- Dérivation d'un symbole
- Complétion d'un parseur

4 Reprise de l'analyse

On peut maintenant :

- réduire petit à petit la pile d'un parser ;
- en répétant la procédure, terminer l'analyse avec un résultat correct.

Que faire des tokens restants ?

L'indentation comme guide

Heuristique générique :

- regarder la colonne à laquelle commence le token
- parmi les parsers produits par la procédure de complétion, chercher ceux à un niveau d'indentation similaire et qui accepteraient de `SHIFT` ce token (sans `REDUCE`).

Niveau d'indentation d'un parser

En regardant l'itemset de l'état courant, prendre la colonne du symbole le moins indenté.
Par exemple, pour :

```
MATCH seq_expr WITH . match_cases
```

Le token susceptible de reprendre après `WITH` doit être plus indenté qu'un des 3 premiers symboles.
Enfin les niveaux d'indentations des parsers doivent être globalement décroissant.

Niveau d’indentation d’un parser

Cette heuristique seule donne de bons résultats en pratique sur OCaml.
Le « parser résilient » est utilisable avec peu de changements.

À l'avenir

- Peut-on annoter des règles d'indentation dans la grammaire ?
- L'édition est interactive : l'historique est une précieuse source d'informations (« ce qui n'a pas changé doit être parsé de la même manière »)
- Autre stratégie à explorer, *reef and abstract islands* :
  Natural and Flexible Error Recovery for Generated Modular Language Environments
  de Jonge, Maartje and Kats, Lennart C. L. and Visser, Eelco and Söderberg, Emmanments