IN 213 **Solution** ENSTA IP PARIS

OCamlyacc — réalisation d'un analyseur syntaxique

Introduction

On considère le langage défini (informellement) ci-après et que nous appellerons «PCF». Il s'agit d'un petit langage fonctionnel similaire à un noyau d'OCaml. C'est un langage interactif, et l'objectif du TP est d'en construire un analyseur syntaxique.

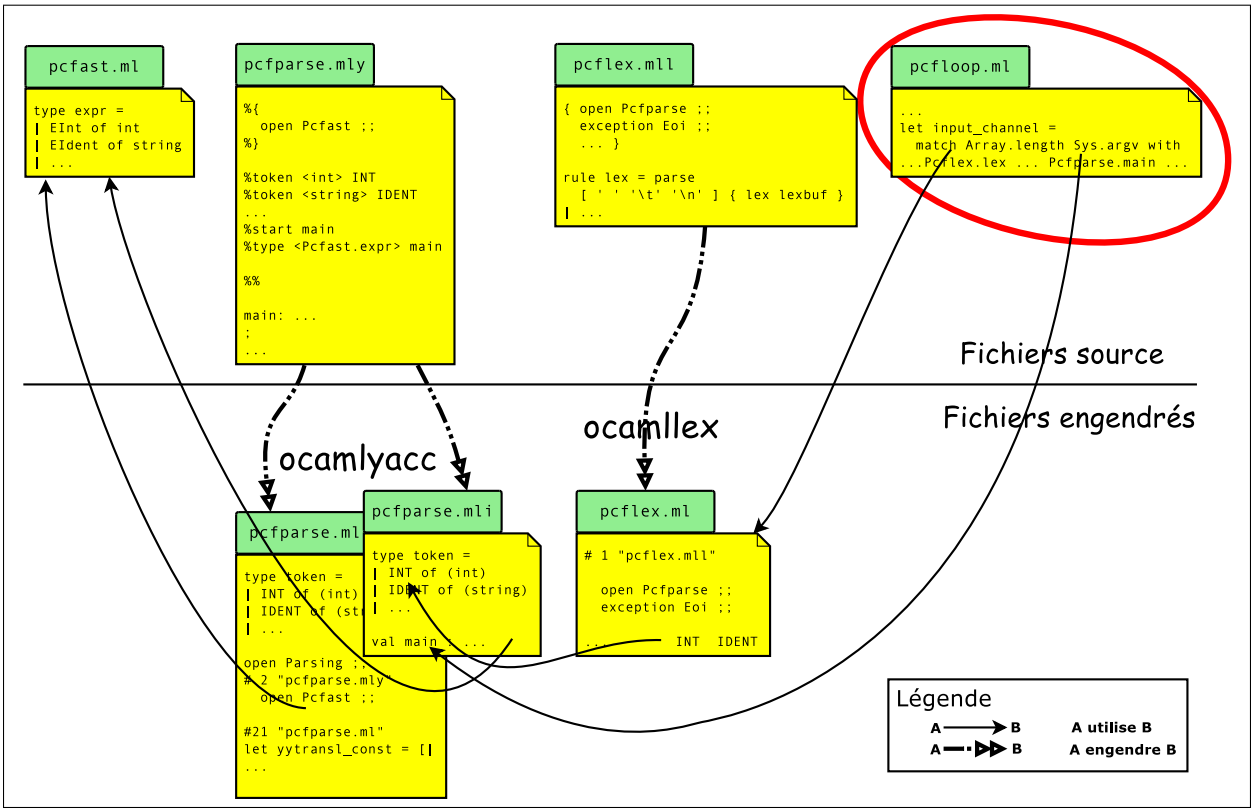

FIGURE 1 – Organisation du programme

L'organisation des fichiers de ce projet est donnée dans le diagramme de la figure 1, où les flèches épaisses indiquent le processus de génération de fichiers, et les autres flèches indiquent quelques dépendances (les flèches vont des utilisations vers les définitions).

Le **programme principal** est le fichier `pcfloop.ml`, qui :

1. imprime un message de bienvenue,
2. crée un `lexbuf` (sur un fichier argument, ou sur l'entrée standard) qui est connecté à l'analyseur lexical de PCF,
3. dans une boucle :

(a) appelle l'analyseur syntaxique en lui passant l'analyseur lexical et le `lexbuf`
(b) imprime (pour votre information) l'expression qu'il vient de lire,
(c) passe à l'expression suivante.

Compiler et recompiler

La commande `make` permet de reconstruire l'exécutable `pcfloop`, après chacune de vos modifications.

Note : La toute première fois, après récupération des fichiers, vous devrez créer un fichier (vide) `.depend` dans le répertoire où vous aurez mis les fichiers. Cela permettra une gestion automatisée des dépendances de compilation (quoi recompiler quand un fichier a été modifié). Vous invoquerez alors `make depend` afin que ces dépendances soient calculées initialement. Lorsque vous modifierez vos fichiers, si vous utilisez une fonction d'une unité de compilation non encore utilisée dans un fichier, la compilation pourra échouer car les dépendances entre fichiers ne seront peut-être pas à jour (en fait, c'est très peu probable car le squelette impose déjà les dépendances nécessaires). Dans ce cas, re-invoquez `make depend`.

Le langage PCF

Un «programme» PCF est une séquence d'expressions séparées par des points-virgules. La grammaire (informelle et incomplète) de PCF peut s'écrire :

```
main : expr SEMICOLON
expr :
  LET IDENT EQUAL expr IN expr
| FUN IDENT ARROW expr
| expr expr
| INT | IDENT | LPAR expr RPAR
| expr PLUS expr | ...
```

où la correspondance entre lexèmes (tokens) et mots-clés est la suivante :

`FUN`	représente	`fun`
`LET`	représente	`let`
`IN`	représente	`in`
...		

(voir `pcflex.mll` pour les détails).

1 Travail à faire

Q1 Dans votre répertoire de travail, exécutez la commande `make`, qui va procéder à la compilation du projet, et créera la commande `pcfloop`, que vous pouvez tester ainsi :

```
$ ./pcfloop
> 0 ;;
Recognized: "A FINIR"
> ^D
Bye bye.
```

Q2 Dans le fichier `pcfast.ml`, examinez le type de données OCaml qui représente les arbres de syntaxe abstraite du langage. Ce fichier définit aussi une fonction d'impression de ces AST, qui est appelée dans le programme principal afin que vous puissiez voir ce que l'analyseur syntaxique a reconnu.

Q3 Observez `pcflex.mll` qui fournit un analyseur lexical pour ce langage, relevez la liste des lexèmes utilisés, et consultez le fichier `pcfparse.mli`, qui définit le type des lexèmes à partir de leurs occurrences dans la première partie du fichier `pcfparse.mly`.

Vous allez devoir enrichir l'analyseur syntaxique dans `pcfparse.mly` pour accepter (successivement) des sous-langages de plus en plus grands, en veillant à produire dans les actions les AST correspondants. À chaque étape, compilez (`make`) et testez votre programme (`./pcfloop`).

Q4 Considérez d'abord le sous-langage :

```
main : expr SEMICOLON | SEMICOLON main
expr : INT | IDENT | LPAR expr RPAR
```

Solution

```
%{
  open Pcfast ;;
%}

%token <int> INT
%token <string> IDENT
%token TRUE FALSE
%token <string> STRING
%token PLUS MINUS MULT DIV EQUAL GREATER SMALLER GREATEREQUAL SMALLEREQUAL
%token LPAR RPAR SEMICOLON
%token LET REC IN FUN ARROW
%token IF THEN ELSE
%left EQUAL GREATER SMALLER GREATEREQUAL SMALLEREQUAL
%left PLUS MINUS
%left MULT DIV

%start main
%type <Pcfast.expr> main

%%

main: expr SEMICOLON { $1 }
    | SEMICOLON main { $2 }
;

/* Grammaire */

expr:
```

```
| INT           { EInt ($1) }
| TRUE          { EBool (true) }
| FALSE         { EBool (false) }
| STRING        { EString ($1) }
| IDENT         { EIdent ($1) }
| LPAR expr RPAR { $2 }
;
```

Q5 Ajoutez les fonctions anonymes «`FUN ... ARROW ...`»

Solution
Il suffit à nouveau de rajouter une production pour le non-terminal `expr` :

```
expr:
| FUN IDENT ARROW expr
        { EFun ($2, $4) }
...
;
```

Q6 Ajoutez les déclarations locales «`LET ... IN ...`». Pour les définitions de fonctions, on aimerait bien bénéficier de la syntaxe OCaml «raccourcie» : `let f = fun x -> fun y -> e` ≡ `let f x y = e`.

Solution
Il suffit de rajouter une production pour le non-terminal `expr` :

```
seqident:
| IDENT seqident { $1 :: $2 }
| /* rien */     { [] }
;

expr:
| LET IDENT seqident EQUAL expr IN expr
                { ELet ($2, (mkfun $3 $5) , $7) }
...
;
```

Vu que `let f x y =` correspond à la liaison de `f` à une fonction de paramètre `x` et de corps étant une fonction (anonyme) de paramètre `y`, on se crée une fonction utilitaire `mkfun` qui prend une liste de paramètres et un corps et qui génère une imbrication d'expressions `EFun` dont la dernière a pour corps celui reçu en argument.

On va mettre cette fonction dans le prélude du fichier `pcfparse.mly`.

```
%{
open Pcfast ;;

let rec mkfun params expr =
  match params with
  | [] -> expr
  | p :: prms -> EFun (p, mkfun prms expr)
;;
%}
```

Q7 Ajoutez les applications de sorte qu'une application «e_1 e_2 e_3» soient traitée comme «(e_1 e_2) e_3». Si vous n'y arrivez pas rapidement, passez à la question suivante (**Q8**).

Solution
La solution naïve consistant à rajouter une production pour le non-terminal `expr` de la façon suivante :

```
expr:
| expr expr      { EApp ($1, $2) }
 ...
;
```

ne fonctionne malheureusement pas car elle génère de nombreux conflits :

```
$ make
ocamlyacc pcfparse.mly
24 shift/reduce conflicts.
```

Il convient de donner plus de structure à notre langage, de contraindre sa grammaire. Si l'on considère l'expression `let f x = x in f 4 ;` comment doit-elle être comprise ?

— `(let f x = x in (f 4))` ?

— `((let f x = x in f) 4)` ?

Nous voyons clairement qu'une production comme celle que nous avons naïvement rajoutée laisse les deux interprétations possibles, d'où la présence de conflits. Il faut empêcher une telle indétermination.

Q8 Nous choisissons (donc) qu'une application est une juxtaposition (une séquence) de constructions «atomiques». Si l'on souhaite utiliser une autre construction dans l'application, il faudra la parenthéser.

Munis de cette indication, reprenez l'ajout de l'application dans la grammaire du langage.

Solution

Nous séparons donc les expressions en 2(-3) groupes :

1. expressions arbitraires `LET`, `FUN`, qui peuvent aussi être des ...
   (a) applications (liste d'applications à une expression «simple»),
2. les expressions «simples», qui peuvent aussi être des expressions arbitraires **parenthésées**.

Notons que les cas expression «identificateur», entier ou autres expressions «simples» tombent dans le cas trivial d'une application avec uniquement une seule expression.

```
main: expr SEMICOLON { $1 }
    | SEMICOLON main { $2 }
;

/* Grammaire */

expr:
| FUN IDENT ARROW expr
        { EFun ($2, $4) }
| LET IDENT seqident EQUAL expr IN expr
                { ELet ($2, (mkfun $3 $5) , $7) }
| application   { $1 }
;

seqident:
| IDENT seqident { $1 :: $2 }
| /* rien */      { [] }
;

application:
| application atom { EApp ($1, $2) }
| atom              { $1 }
;

atom:
| INT           { EInt ($1) }
| TRUE          { EBool (true) }
| FALSE         { EBool (false) }
| STRING        { EString ($1) }
| IDENT         { EIdent ($1) }
| LPAR expr RPAR { $2 }
;
```

2 Si vous en avez le temps

Enrichissez votre langage avec des expressions arithmétiques et booléennes, ainsi que les déclarations récursives afin d'être en mesure de reconnaître avec succès le programme PCF contenu dans le fichier `fact.pcf`.

Indication : pour l'ajout des expressions arithmétiques, stratifiez une fois de plus votre grammaire en introduisant un non-terminal `arith_expr` qui peut être une expression avec un opérateur binaire, ou bien une `application`.

Solution

```
expr:
  ...
```

```
| arith_expr
;

arith_expr:
| arith_expr EQUAL arith_expr
| ...
| application
;
```