Extensibilité syntaxique et sémantique : macros C++ , OCaml PPX, décorateurs pythons et MetaOCaml

Barry Alpha Volte--Vieira Philippe

M2S2 2022

I Rappels sur la compilation

Les compilateurs sont dans la plupart des cas structurés avec une première partie nommée *lexer* qui reconnaît les mots-clefs et valeurs présentes dans le fichier source. Ces informations sont envoyées au *parser* qui construit l'arbre syntaxique du programme grâce à une structure d'arbre de syntaxe abstraite (AST). S'il n'y a pas d'erreur la traduction du code source en exécutable peut opérer : le programme, sous forme d'AST, est envoyé aux autres modules du compilateur qui produiront l'exécutable associé au programme, ou alors renverront une erreur.

Une extension syntaxique ou sémantique étend en pratique le comportement d'un compilateur, leur traitement se fait donc avant la traduction à proprement parler. On parle alors de *préprocesseurs* (processus précédant la compilation).

II Macros C++

En C++ les macros permettent d'associer un nom à un bout de code. Le nom de la macro pourra ensuite être utilisé dans le programme comme une variable ou une fonction déjà déclarée. Ainsi toutes les occurrences du nom seront remplacées par le bout de code correspondant juste avant la compilation (avant le *parsing*). Cette substitution est effectuée par le préprocesseur.

II.1 Le préprocesseur CPP

Le préprocesseur CPP est un programme appelé automatiquement par le compilateur tout au début de la compilation. Il permet d'effectuer un certains nombre de transformations textuelles comme la suppression des commentaires. Il permet aussi d'effectuer des tâches en suivant des ordres qui lui sont destinées à travers des directives de préprocesseur. Il s'agit entre autre de l'inclusion de fichiers, de la compilation conditionnelle et de l'expansion de macros. Il est possible de consulter la sortie du préprocesseur en utilisant la commande `cpp` ou en passant l'option `-E` à `gcc`.

II.2 Les macros

Une macro est définie à l'aide de la directive `#define`. On distingue deux types de macros : les Object-like macros et les Function-like macros.

- **Les Object-like macros** sont de la forme `#define name body` (où `name` est le nom de la macro et `body` une séquence de jetons). Ils sont souvent utilisés pour donner des noms symboliques aux constantes numériques.
- **Les Function-like macros** sont de la forme `#define name() body` (comme les Object-like macros en ajoutant une paire de parenthèses juste après le nom de la macro). Leur utilisation ressemble à des appels de fonctions avec des arguments non-typés.

Exemple 1 : analyse séquentielle

```
1 foo = X;
2 #define X 4
3 bar = X;
```

Le préprocesseur analyse le fichier source de manière séquentielle.
Avec ce code le compilateur verra le code ci-contre :

```
1 foo = X;
2 bar = 4;
```

Exemple2 : attention aux parenthèses

```
#define SUM(x, y) x + y // une macro qui fait la somme de deux nombres
sum = SUM(3, 5) * 2;
```

Avec ce code le resultat attendu est `sum = 8 * 2`. Mais ce que le compilateur verra est `sum = 3 + 5 * 2`. Et du fait de la priorité de l'opérateur `*` sur l'opérateur `+` on aura `sum = 13`. Pour y remédier il faudra ajouter des parenthèses : `#define SUM(x, y) ((x) + (y))`

Exemple3 : Auto référence

```
#define foo (4 + foo)
x = foo;
```

Lorsque le préprocesseur développe une macro, s'il rencontre le nom de la macro dans son corps celui-ci ne sera pas développé à nouveau. Ce qui permet d'éviter des définitions récursives infinies. Ainsi avec le code c-dessus on aura : `x = 4 + foo;`.

II.3 Critiques

Si les macros nous permettent d'économiser des appels de fonction en copiant le code à l'endroit où on en a besoin, elles présentent plus d'inconvénients. Les macros ne sont pas typés, donc pourraient être source d'erreur.

Une macro doit tenir sur une seule ligne. Mais pour la lisibilité il est possible d'utiliser des "backslash" pour écrire la macro sur plusieurs lignes. Ces lignes supplémentaires seront supprimées par le préprocesseur. Ce qui peut engendrer des surprises sur les numéros de lignes dans les messages d'erreur.

Les arguments d'une macros sont évalués à chaque fois qu'ils sont utilisés dans son corps. Ce qui peut engendrer un problème de performance mais aussi des effets de bord indésirables.

Il y a d'autre inconvénients liés à l'utilisation des macros. Scott Meyers nous dit dans son livre *Effective C++ Third Edition* : "prefer the compiler to the preprocessor". Il s'agit de préférer la classe de stockage `const` (pour les constantes) et les fonctions `inline` (pour les fonction) à la place des macros.

III OCaml PPX

Le compilateur OCaml intègre la gestion des extension : on indique dans le fichier source quels bouts de code doivent être traités par les extensions via les symboles `@` et `%`. Le compilateur intègre ces informations dans l'AST, celui-ci est d'abord transformé par les extensions en code OCaml valide, puis traduit par le compilateur.

Le passage de l'AST aux extensions permet une grande liberté de comportement : les extensions peuvent par exemple intervertir toutes les branches `then` et `else` du programme !

Ces extensions sont nommées les PreProcessor eXtensions, d'où OCaml PPX. Nous les appellerons PPX par la suite.

Il existe deux types de nœuds insérables dans l'AST.

— **Les extensions** *remplacent* un nœud OCaml. On utilise la syntaxe `[%ext payload]` où `ext` est le nom de l'extension et `payload` du code traité par celle-ci.

— **Les attributs** *s'accrochent* à un nœud OCaml. On utilise la syntaxe `[@attr opt]` où `attr` est le nom de l'extension et `opt` une suite d'options.

Lorsqu'on utilise une extension on devra donc obligatoirement spécifier au compilateur quelles PPX utiliser, contrairement aux dérivateurs qui peuvent être simplement ignorés.

Ainsi, le compilateur natif d'OCaml arrive à reconnaître le code suivant qui doit être traité par la PPX `ext` :

```
let e = [%ext function Some v -> expr1 | None -> expr2]
let a = 12 [@ext opt]
```

Les nœuds que les PPX ajoutent ont une syntaxe différente selon s'ils concernent une expression simple ou une structure. Pour remplacer une expression ou s'y accrocher la syntaxe est celle vu ci-dessus.

En ce qui concerne les modules[1] on utilise `[%%ext payload]` et `[@@attr opt]`. Enfin, les attributs peuvent être flottants (en *toplevel*) en utilisant la syntaxe `[@@@attr opt]`.

```
let a = 12
[@attr opt] (* associé au 12 *)
[@@attr2 opt] (* ass. au let *)
[@@@attr3 opt] (* flottant *)
```

III.1 Dérivateurs pour une calculatrice

Prenons l'exemple d'une calculatrice basée sur le type ci-contre. Nous utilisons la fonction `show` pour afficher une instance de ce type.

Imaginons que l'on aie un code bien plus lourd et complexe. Si on modifie le type en ajoutant `Div of int*int` il faut alors modifier chaque fonction, même les plus simples comme `show`. Nous pouvons à la place utiliser les *dérivateurs* du paquet `ppx_deriving`.

```
type t = Add of int * int
  | Mul of int * int
let show = function
  | Add (a, b) ->
    "Add ("^(Int.to_string a)^
       ", "^(Int.to_string b)^")"
  | Mul (a, b) ->
    "Mul ("^(Int.to_string a)^
       ", "^(Int.to_string b)^")"
```

```
type t = ...[@@deriving show] (* show inséré automatiquement par ppx_deriving *)
```

III.2 Extensions pour une calculatrice

Une calculatrice lit des opérations en entrée, depuis le type `string`. Pour ce faire on peut utiliser une fonction `s_match` faisant appel aux fonctions du module `Str` et utilisant les expressions régulières. Mais on aurait alors des `match ... with` imbriqués (un par forme attendue) et il faudrait récupérer les valeurs spécifiées via une structure comme une liste.

La PPX `ppx_regexp` nous permet de faire une distinction de cas directement sur les expressions, et de récupérer les arguments spécifiés.

```
let read =
  let to_int = Stdlib.int_of_string in
  (function%pcre
    | {|\s*(?<i>[0-9]+)\s*\+\s*(?<j>[0-9]+)\s*|} -> Add (to_int i, to_int j)
    | {|\s*(?<i>[0-9]+)\s*\*\s*(?<j>[0-9]+)\s*|} -> Mul (to_int i, to_int j)
    | _ -> failwith "Unknown operation"
  )
```

IV Décorateurs Python

Un décorateur python est un `Callable`, un objet qui peut être appelé avec la syntaxe `nom_objet()`, qui prend en argument une fonction (la fonction à décorer) et retourne un `Callable`. Le principe est similaire au patron de conception Decorator du GOF. Il permet d'ajouter dynamiquement une nouvelle fonctionnalité à une fonction. Il permet aussi de factoriser une logique ou besoin commun à plusieurs fonctions.

Rappelons qu'en Python les fonctions sont des objets de première classe.

1. La documentation est floue sur le sujet, il semble que l'on puisse utiliser les deux premières syntaxes pour les modules et leur types.

Nous définissons un décorateur `is_logged` qui permet d'appeler la fonction `show_profile` seulement si l'utilisateur est connecté (`logged` vaut `True`) en utilisant l'annotation `@is_logged`. Cette syntaxe est un raccourci ou sucre syntaxique pour le code `is_logged(show_profile)()`.

Il peut donc être pratique de pouvoir indiquer dès la définition de la fonction `show_profile` qu'elle n'est appelée seulement si l'utilisateur est bien connecté. Et nous pouvons appliquer cette contrainte à bien d'autres fonctions. En d'autres termes nous modifions le comportement de ces fonctions sans leur ajouter du code.

```
def is_logged(func):
  def wrapper():
    if logged :
      return func()
    print("You must be logged!")
  return wrapper

@is_logged
def show_profile():
  print("Your profile")
```

V MetaOCaml

MetaOCaml est une extension d'OCaml visant à permettre le *Multi-Stage programming*. L'objectif est, notamment, de "dérouler" certains appels de fonction lors de la compilation pour optimiser l'exécution du programme.

La dénomination MetaOCaml peut s'expliquer par la génération de code : pour une fonction `f` on spécifie à MetaOCaml comment la "dérouler" (pour une partie des arguments donnée), ce qui équivaut à générer des fonctions correspondant à des applications partielles de `f`. Notre programme générant des programmes (les applications partielles de `f`), on parle bien de méta-programmation.

Le code annoté avec `.<e>.` indique que `e` est disponible (calculé), et `.~e` indique que `e` sera calculé après (par exemple, au *runtime*).

Pour une fonction calculant la puissance d'un nombre, on peut indiquer à quels moments les résultats sont produits grâce aux annotations de MetaOCaml.

En commentaire (la réponse de MetaOCaml en toplevel) nous pouvons voir que la spécialisation de `spower` avec `n` valant `7` est déroulée par MetaOCaml pour n'avoir que les multiplications sans les appels récursifs.

```
let square x = x * x
let rec spower n x =
  if n=0 then .<1>.
  else if n mod 2 = 0
  then .<square .~(spower (n/2) x)>.
  else .<.~x * .~(spower (n-1) x)>.
(* val spower : int -> int code ->
     int code = <fun> *)
```

```
let spower7_code = (* spécialisation de spower pour n=7 *)
  .<fun x -> .~(spower 7 .<x>.)>.
(* val spower7_code : (int -> int) code = .<
     fun x_1 -> x_1 * ((* CSP square *) (x_1 * ((* CSP square *) (x_1 * 1))))>.
*)
let spower7 = Runnative.run spower7_code
(* val spower7 : int -> int = <fun> *)
```

Pour `spower7_code` on a bien que on a bien `x -> x*(x*(x*1)²)² = x*(x³)² = x^7`

VI Conclusion

Du simple sucre syntaxique jusqu'à la personnalisation des compilateurs, les extensions de langages introduisent une nouvelle façon d'appréhender les langages de programmation, en élargissant les cadres définis par ceux-ci.

Bibliographie

[1] The C preprocessor. https://gcc.gnu.org/onlinedocs/cpp/.

[2] Cours de C/C++, le préprocesseur C. https://cpp.developpez.com/cours/cpp/?page=page_7.

[3] Documentation : Decorators for functions and methods. https://www.python.org/dev/peps/pep-0318/.

[4] Dépôt github de ppxderiving. https://github.com/ocaml-ppx/ppx_deriving.

[5] Metaocaml – an OCaml dialect for multi-stage programming. https://okmij.org/ftp/ML/MetaOCaml.html.

[6] Perry Metzger. A guide to PreProcessor eXtensions. https://ocamlverse.github.io/content/ppx.html.

[7] Nathan Rebours. An introduction to OCaml PPX ecosystem. https://tarides.com/blog/2019-05-09-an-introduction-to-ocaml-ppx-ecosystem, May 2019.