

Métaprogrammation

Vincent Archambault-B
IFT 2035 - Université de Montréal



Ce document est dédié au domaine public via [CCO](#)

Pour obtenir le code source de ce document

- ▶ <https://github.com/archambaultv/IFT2035-UdeM>
- ▶ vincent.archambault-bouffard@umontreal.ca

Plan du cours

- ▶ Définition
- ▶ Approche textuelle
- ▶ Approche avec l'arbre de syntaxe abstraite (ASA)
- ▶ Hygiène
- ▶ Nouvelle syntaxe

Métaprogrammation

- ▶ Désigne l'écriture de programmes qui manipulent des données décrivant elles-mêmes des programmes (Wikipédia).

Métabrogrammation textuelle

Métaprogrammation textuelle

- ▶ Programme représenté par une chaîne de caractères
- ▶ Utilisée en C
- ▶ Des fois appelée « preprocessor »

```
#define MAX(a,b) a > b ? a : b;

int main(){
    int a = MAX(5,4); // a = 5
    return 0;
}
```

Macro C

Métaprogrammation textuelle

```
#define MAX(a,b) a > b ? a : b;

int main(){
    int a = MAX(5,4);
    return 0;
}
```

Code avec macro C

```
#define MAX(a,b) a > b ? a : b;

int main(){
    int a = 5 > 4 ? 5 : 4;
    return 0;
}
```

Code une fois la macro expansée

Métaprogrammation textuelle

```
#define DETERMINANT(a,b,c) \  
    b * b - 4 * a * c;  
  
int main(){  
    int delta = DETERMINANT(4,10,5);  
    return 0;  
}
```

Code avec macro C

```
#define DETERMINANT(a,b,c) \  
    b * b - 4 * a * c;  
  
int main(){  
    int delta = 10 * 10 - 4 * 4 * 5;  
    return 0;  
}
```

Code une fois la macro expansée

Métaprogrammation textuelle

- ▶ Une substitution textuelle peut parfois donner lieu à des problèmes au niveau de l'analyse syntaxique

Problème d'analyse syntaxique

```
#define MAX(a,b) a > b ? a : b;

int main(){
    int a = 1 - MAX(5,4);
    return 0;
}
```

Code avec macro C

```
#define MAX(a,b) a > b ? a : b;

int main(){
    int a = 1 - 5 > 4 ? 5 : 4;
    return 0;
}
```

Code une fois la macro expansée

Problème d'analyse syntaxique

```
#define MAX(a,b) a > b ? a : b;

int main(){
    int a = 1 - MAX(5,4);
    return 0;
}
```

Code avec macro C

```
#define MAX(a,b) a > b ? a : b;

int main(){
    int a = (1 - 5) > 4 ? 5 : 4;
    return 0;
}
```

Code une fois la macro expansée

Problème d'analyse syntaxique

```
#define DETERMINANT(a,b,c) \  
    b * b - 4 * a * c;  
  
int main(){  
    int delta = 4 * DETERMINANT(4,10,5);  
    return 0;  
}
```

Code avec macro C

```
#define DETERMINANT(a,b,c) \  
    b * b - 4 * a * c;  
  
int main(){  
    int delta = 4 * 10 * 10 - 4 * 4 * 5;  
    return 0;  
}
```

Code une fois la macro expansée

Problème d'analyse syntaxique

```
#define DETERMINANT(a,b,c) \  
    b * b - 4 * a * c;  
  
int main(){  
    int delta = 4 * DETERMINANT(4,10,5);  
    return 0;  
}
```

Code avec macro C

```
#define DETERMINANT(a,b,c) \  
    b * b - 4 * a * c;  
  
int main(){  
    int delta = (4 * 10 * 10) - 4 * 4 * 5;  
    return 0;  
}
```

Code une fois la macro expansée

Solution au problème d'analyse syntaxique : Mettre des parenthèses

```
#define MAX(a,b) (a > b ? a : b);

int main(){
    int a = 1 - MAX(5,4);
    return 0;
}
```

Il faut ajouter des parenthèses

```
#define MAX(a,b) (a > b ? a : b);

int main(){
    int a = 1 - (5 > 4 ? 5 : 4);
    return 0;
}
```

Code une fois la macro expansée

Solution au problème d'analyse syntaxique : Mettre des parenthèses

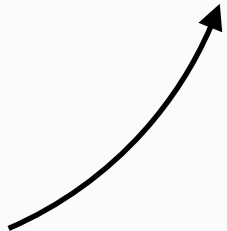
```
#define MAX(a,b) (a > b ? a : b);

int main(){
    int a = 1 - MAX(2,1 > 3);
    return 0;
}
```

Il faut ajouter des parenthèses

```
#define MAX(a,b) (a > b ? a : b);

int main(){
    int a = 1 - (2 > 1 > 3 ? 2 : 1 > 3);
    return 0;
}
```

Erreur 

Code une fois la macro expansée

Solution au problème d'analyse syntaxique : Mettre beaucoup de parenthèses

```
#define MAX(a,b) \  
    ((a) > (b) ? (a) : (b));  
  
int main(){  
    int a = 1 - MAX(2,1 > 3);  
    return 0;  
}
```

Il faut ajouter des parenthèses **PARTOUT**

```
#define MAX(a,b) \  
    ((a) > (b) ? (a) : (b));  
  
int main(){  
    int a = 1 - ((2) > (1 > 3) ? (2) :  
                (1 > 3));  
    return 0;  
}
```

Code une fois la macro expansée

Métabrogrammation sur l'ASA

Métaprogrammation sur l'ASA

- ▶ Une façon plus efficace de régler le problème d'analyse syntaxique est de manipuler non pas des chaînes de caractères mais des arbres de syntaxe abstraite
- ▶ Tous les systèmes de métaprogrammation modernes sont conçus ainsi (LISP, Scheme, Haskell, Scala, Python, etc.)

LISP et Scheme

- ▶ LISP et Scheme utilisent des S-expressions comme syntaxe
- ▶ Les S-expressions sont équivalentes à l'ASA
- ▶ Manipuler l'ASA revient à manipuler des S-expressions

Macros LISP (en Racket)

```
#lang racket

(require compatibility/defmacro)

(define-macro (max a b)
  (list 'if (list '> a b) a b))

(define a (- 1 (max 5 4)))
```

Racket : Utiliser l'ASA pour représenter le code

```
#lang racket

(require compatibility/defmacro)

(define-macro (max a b)
  (list 'if (list '> a b) a b))

(define a (- 1 (if (> 5 4) 5 4)))
```

Code une fois la macro expansée

Macros LISP, syntaxe spéciale

```
#lang racket

(require compatibility/defmacro)

(define-macro (max a b)
  `(if (> ,a ,b) ,a ,b)))

(define a (- 1 (max 5 4)))
```

Une macro LISP peut analyser ses arguments plus facilement que s'ils étaient une chaîne de caractères

```
#lang racket

(require compatibility/defmacro)

(define-macro (smart-max a b)
  (if (equal? a b)
      a
      (list 'if (list '> a b) a b)))

(define b (smart-max 5 5))
```

Racket : Utiliser l'ASA pour représenter le code

```
#lang racket

(require compatibility/defmacro)

(define-macro (smart-max a b)
  (if (equal? a b)
      a
      (list 'if (list '> a b) a b)))

(define b 5)
```

Code une fois la macro expansée

Métaprogrammation sur l'ASA

- ▶ Règle le problème d'analyse syntaxique
- ▶ La méta-programmation reste toutefois moins intuitive car :
 - ▶ Passage par nom
 - ▶ Porté dynamique

Passage par nom

```
#lang racket

(require compatibility/defmacro)

(define-macro (max a b)
  (list 'if (list '> a b) a b))

(define c 0)
(define d (max (begin (set! c (+ c 1))
                     5)
               4))
```

Racket : Utiliser l'ASA pour représenter le code

```
#lang racket

(require compatibility/defmacro)

(define-macro (max a b)
  (list 'if (list '> a b) a b))

(define c 0)
(define d
  (if (> (begin (set! c (+ c 1)) 5)
      4)
      (begin (set! c (+ c 1)) 5)
      4)) ; c vaut 2 après évaluation
```

Code une fois la macro expansée

Passage par nom : utiliser des let temporaires

```
#lang racket

(require compatibility/defmacro)

(define-macro (max a b)
  `(let ([tmpA ,a]
         [tmpB ,b])
      (if (> tmpA tmpB) tmpA tmpB)))
```

Portée dynamique (capture)

```
#lang racket

(require compatibility/defmacro)

(define-macro (max a b)
  `(let* ([tmpA ,a]
          [tmpB ,b])
     (if (> tmpA tmpB) tmpA tmpB)))

(define f (let ([tmpA 4])
            (max 3 tmpA)))
```

Racket : Utiliser l'ASA pour représenter le code

```
#lang racket

(require compatibility/defmacro)

(define-macro (max a b)
  (list 'if (list '> a b) a b))

(define f (let ([tmpA 4])
            `(let* ([tmpA 3]
                    [tmpB tmpA])
               (if (> tmpA tmpB) tmpA tmpB))))
```

Code une fois la macro expansée

Portée dynamique (capture) : utiliser gensym

```
#lang racket

(require compatibility/defmacro)

(define-macro (max3 a b)
  (let ([tmpA (gensym)]
        [tmpB (gensym)])
    `(let* ([,tmpA ,a]
            [,tmpB ,b])
      (if (> ,tmpA ,tmpB)
          ,tmpA
          ,tmpB))))

(define f (let ([tmpA 4])
            (max 3 tmpA)))
```

Racket : Utiliser l'ASA pour représenter le code

```
#lang racket

(require compatibility/defmacro)

(define-macro (max a b)
  (list 'if (list '> a b) a b))

(define f (let ([tmpA 4])
  `(let* ([tmpA_01 3]
          [tmpB_01 tmpA])
    (if (> tmpA_01 tmpB_01)
        tmpA_01
        tmpB_01)))
```

Code une fois la macro expansée

Portée dynamique (référence)

```
#lang racket

(require compatibility/defmacro)

(define-macro (max a b)
  (list 'if (list '> a b) a b))

(define f (let ([> (lambda (x y) #f)])
  (max 5 4)))
```

Racket : Utiliser l'ASA pour représenter le code

```
#lang racket

(require compatibility/defmacro)

(define-macro (max a b)
  (list 'if (list '> a b) a b))

(define f (let ([> (lambda (x y) #f)])
  (if (> 5 4) 5 4)))
```

Code une fois la macro expansée

Portée dynamique (référence) : renommage par le compilateur

```
#lang racket

(require compatibility/defmacro)

(define-macro (max a b)
  (list 'if (list '> a b) a b))

(define f
  (let ([>_1 (lambda (x y) #f)])
    (max 5 4)))
```

Racket : Utiliser l'ASA pour représenter le code

```
#lang racket

(require compatibility/defmacro)

(define-macro (max a b)
  (list 'if (list '> a b) a b))

(define f
  (let ([>_1 (lambda (x y) #f)])
    (if (> 5 4) 5 4)))
```

Code une fois la macro expansée

Hygiène

Hygiène

- ▶ Le problème de la portée dynamique (capture et référence) est connu sous le nom du problème d'hygiène
- ▶ Est-il possible d'avoir une portée « statique » pour les macros ?
- ▶ Oui, mais cela requiert beaucoup plus de travail pour le compilateur et le concepteur du langage

Hygiène : Solution

- ▶ Automatiser les gensym
- ▶ Renommer toutes les variables automatiquement
- ▶ Malgré cela, il existe encore certains cas problématiques
- ▶ L'hygiène est un domaine de recherche toujours actif en 2017

Hygiène : Solution

- ▶ Racket est hygiénique par défaut
(sauf si l'on utilise `(require compatibility/defmacro)`)
- ▶ Pour en savoir plus sur comment implanter l'hygiène voir l'article [Binding as Sets of Scopes](#) de Matthew Flatt

Nouvelle syntaxe

Nouvelle syntaxe

- Les macros permettent d'ajouter de nouvelle syntaxe

Nouvelle syntaxe

```
(if (equal? note "A")
    (display "Très bien")
    (if (equal? note "B")
        (display "Bien")
        (if (equal? note "C")
            (display "Passable")
            (if (equal? note "D")
                (display "Passable")
                (display "Échec")))))
```

Racket avec le if

```
(case note
  [("A") (display "Très bien")]
  [("B") (display "Bien")]
  [("C" "D") (display "Passable")]
  [else (display "Échec")])
```

Racket avec une macro case

Macro case (Racket)

```
(define-syntax case
  (syntax-rules (else)
    ((case (key ...)
      clauses ...)
      (let ((atom-key (key ...)))
        (case atom-key clauses ...)))
    ((case key
      (else result1 result2 ...))
      (begin result1 result2 ...))
    ((case key
      ((atoms ...) result1 result2 ...))
      (if (memv key '(atoms ...))
          (begin result1 result2 ...)))
    ((case key
      ((atoms ...) result1 result2 ...)
      clause clauses ...)
      (if (memv key '(atoms ...))
          (begin result1 result2 ...)
          (case key clause clauses ...)))))
```