



# Interprétation et compilation

## Chapitre 3 Analyse sémantique



Pablo Rauzy <pr@up8.edu>  
pablo.rauzy.name/teaching/ic

# Analyse sémantique

---

- ▶ En compilation, l'*analyse sémantique* est l'étape qui suit l'*analyse syntaxique*.
- ▶ On peut la considérer comme la dernière partie du “front end” du compilateur, puisque c'est potentiellement la dernière qui est spécifique au langage source.
- ▶ Le rôle est de comprendre et vérifier le *sens* du code source, qui est défini par la sémantique du langage.

- ▶ Le but de l'analyse sémantique est de produire un *arbre de syntaxe abstraite* (AST) sémantiquement valide.
- ▶ Cela signifie qu'on doit pouvoir interpréter ou compiler le code représenté par cet AST sans se soucier de ce qui a déjà été vérifié.
- ▶ Selon les langages, ces vérifications sont plus ou moins importantes.

## Vocabulaire : “statique” vs “dynamique”

- ▶ On dit que quelque chose est *statique* quand on peut le déterminer juste à partir du code source, sans avoir besoin d'exécution.
- ▶ On dit que quelque chose est *dynamique* si il ne peut être déterminé qu'à l'exécution.
- ▶ Au moment de la compilation, on est donc sûr de l'*analyse statique*.

→ Quels types de vérifications peuvent être réalisés lors de l'analyse sémantique ?

- Quels types de vérifications peuvent être réalisés lors de l'analyse sémantique ?
- ▶ Le minimum est de vérifier l'utilisation des noms.
    - La façon correcte de le faire dépend du langage.

- Quels types de vérifications peuvent être réalisés lors de l'analyse sémantique ?
- ▶ Le minimum est de vérifier l'utilisation des noms.
    - La façon correcte de le faire dépend du langage.
  - ▶ Le reste de l'analyse sémantique consiste essentiellement au *typage*.
    - Cette étape peut être rudimentaire ou au contraire extrêmement poussée.



- ▶ On appelle la *portée* (*scope* en anglais) d'un nom l'ensemble des points du code source où ce nom est accessible.
- ▶ Vous connaissez déjà le concept :
  - vous avez entendu parler de variables locales et de variables globales,
  - vous avez probablement déjà utilisé les politiques d'accès des attributs en programmation orientée objet.

# Variables libres

- ▶ Dans un morceau de code, une variable est dite *libre* si elle n'est pas locale.
  - Exemple : une variable utilisée dans une fonction mais qui ne fait pas partie de ses arguments ni des variables créées localement dans cette fonction.

# Variables liées

- ▶ Dans un morceau de code, une variable est dite *liée* si elle n'est pas libre.

## Exemple

► Regardons ce code :

- ```
(define x 42)
(define (foo a)
  (define b (+ a v))
  (define c
    (let ((d (* b w))
          (e (- a x)))
      (define f (/ d y))
      (* f z)))
  (+ b c))
(foo (+ (* a x x) (* b x) c))
```

► Quelles sont les variables libres ?

## Exemple

► Regardons ce code :

- ```
(define x 42)
(define (foo a)
  (define b (+ a v))
  (define c
    (let ((d (* b w))
          (e (- a x)))
      (define f (/ d y))
      (* f z)))
  (+ b c))
(foo (+ (* a x x) (* b x) c))
```

► Quelles sont les variables libres ?

## Exemple

► Regardons ce code :

- ```
(define x 42)
(define (foo a)
  (define b (+ a v))
  (define c
    (let ((d (* b w))
          (e (- a x)))
      (define f (/ d y))
      (* f z)))
  (+ b c))
(foo (+ (* a x x) (* b x) c))
```

► Quelles sont les variables liées ?

## Exemple

► Regardons ce code :

- ```
(define x 42)
(define (foo a)
  (define b (+ a v))
  (define c
    (let ((d (* b w))
          (e (- a x)))
      (define f (/ d y))
      (* f z)))
  (+ b c))
(foo (+ (* a x x) (* b x) c))
```

► Quelles sont les variables liées ?

## Exemple

► Regardons ce code :

- ```
(define x 42)
(define (foo a)
  (define b (+ a v))
  (define c
    (let ((d (* b w))
          (e (- a x)))
      (define f (/ d y))
      (* f z)))
  (+ b c))
(foo (+ (* a x x) (* b x) c))
```

► Quelle est la portée de a ?



## Exemple

► Regardons ce code :

- ```
(define x 42)
(define (foo a)
  (define b (+ a v))
  (define c
    (let ((d (* b w))
          (e (- a x)))
      (define f (/ d y))
      (* f z)))
  (+ b c))
(foo (+ (* a x x) (* b x) c))
```

► Quelle est la portée de a ?

## Exemple

► Regardons ce code :

- ```
(define x 42)
(define (foo a)
  (define b (+ a v))
  (define c
    (let ((d (* b w))
          (e (- a x)))
      (define f (/ d y))
      (* f z)))
  (+ b c))
(foo (+ (* a x x) (* b x) c))
```

► Quelle est la portée de **b** ?

## Exemple

► Regardons ce code :

- ```
(define x 42)
(define (foo a)
  (define b (+ a v))
  (define c
    (let ((d (* b w))
          (e (- a x)))
      (define f (/ d y))
      (* f z)))
  (+ b c))
(foo (+ (* a x x) (* b x) c))
```

► Quelle est la portée de **b** ?

## Exemple

► Regardons ce code :

- ```
(define x 42)
(define (foo a)
  (define b (+ a v))
  (define c
    (let ((d (* b w))
          (e (- a x)))
      (define f (/ d y))
      (* f z)))
  (+ b c))
(foo (+ (* a x x) (* b x) c))
```

► Quelle est la portée de c ?

## Exemple

► Regardons ce code :

- ```
(define x 42)
(define (foo a)
  (define b (+ a v))
  (define c
    (let ((d (* b w))
          (e (- a x)))
      (define f (/ d y))
      (* f z)))
  (+ b c))
(foo (+ (* a x x) (* b x) c))
```

► Quelle est la portée de c ?

## Exemple

► Regardons ce code :

- ```
(define x 42)
(define (foo a)
  (define b (+ a v))
  (define c
    (let ((d (* b w))
          (e (- a x)))
      (define f (/ d y))
      (* f z)))
  (+ b c))
(foo (+ (* a x x) (* b x) c))
```

► Quelle est la portée de **d** ?

## Exemple

► Regardons ce code :

- ```
(define x 42)
(define (foo a)
  (define b (+ a v))
  (define c
    (let ((d (* b w))
          (e (- a x)))
      (define f (/ d y))
      (* f z)))
  (+ b c))
(foo (+ (* a x x) (* b x) c))
```

► Quelle est la portée de `d` ?

## Exemple

► Regardons ce code :

- ```
(define x 42)
(define (foo a)
  (define b (+ a v))
  (define c
    (let ((d (* b w))
          (e (- a x)))
      (define f (/ d y))
      (* f z)))
  (+ b c))
(foo (+ (* a x x) (* b x) c))
```

► Quelle est la portée de `e` ?



## Exemple

► Regardons ce code :

```
• (define x 42)
  (define (foo a)
    (define b (+ a v))
    (define c
      (let ((d (* b w))
            (e (- a x)))
        (define f (/ d y))
        (* f z)))
    (+ b c))
  (foo (+ (* a x x) (* b x) c))
```

► Quelle est la portée de `e` ?

## Exemple

► Regardons ce code :

- ```
(define x 42)
(define (foo a)
  (define b (+ a v))
  (define c
    (let ((d (* b w))
          (e (- a x)))
      (define f (/ d y))
      (* f z)))
  (+ b c))
(foo (+ (* a x x) (* b x) c))
```

► Quelle est la portée de f ?

## Exemple

► Regardons ce code :

- ```
(define x 42)
(define (foo a)
  (define b (+ a v))
  (define c
    (let ((d (* b w))
          (e (- a x)))
      (define f (/ d y))
      (* f z)))
  (+ b c))
(foo (+ (* a x x) (* b x) c))
```

► Quelle est la portée de f ?

## Exemple

► Regardons ce code :

- ```
(define x 42)
(define (foo a)
  (define b (+ a v))
  (define c
    (let ((d (* b w))
          (e (- a x)))
      (define f (/ d y))
      (* f z)))
  (+ b c))
(foo (+ (* a x x) (* b x) c))
```

► Quelle est la portée de `foo` ?

## Exemple

► Regardons ce code :

- ```
(define x 42)
(define (foo a)
  (define b (+ a v))
  (define c
    (let ((d (* b w))
          (e (- a x)))
      (define f (/ d y))
      (* f z)))
  (+ b c))
(foo (+ (* a x x) (* b x) c))
```

► Quelle est la portée de `foo` ?

## Exemple

► Regardons ce code :

- ```
(define x 42)
(define (foo a)
  (define b (+ a v))
  (define c
    (let ((d (* b w))
          (e (- a x)))
      (define f (/ d y))
      (* f z)))
  (+ b c))
(foo (+ (* a x x) (* b x) c))
```

► Quelle est la portée de x ?

## Exemple

► Regardons ce code :

- `(define x 42)`  
`(define (foo a)`  
  `(define b (+ a v))`  
  `(define c`  
    `(let ((d (* b w))`  
      `(e (- a x)))`  
    `(define f (/ d y))`  
    `(* f z)))`  
  `(+ b c))`  
`(foo (+ (* a x x) (* b x) c))`

► Quelle est la portée de x ?

# Portée dynamique

- ▶ On parle de *portée dynamique* des variables si les variables libres d'une fonction prennent leur valeur dans l'*environnement dynamique*.
- ▶ L'environnement dynamique est celui de l'exécution.
- ▶ Dans le code il correspond au contexte de l'appel de la fonction.



## Exemple : Bash

► Considérons la session shell Bash suivante :

- ```
$ x=42
$ function foo() { echo $x; }
$ foo
42
$ x=13
$ foo
13
```

## Exemple : Emacs Lisp

► Considérons le code Emacs Lisp suivant :

- `(defvar x "coucou")`

```
(defun foo ()  
  (message x))
```

```
(foo) ;; affiche "coucou"
```

```
(let ((x "salut"))  
  (foo)) ;; affiche "salut"
```

- ▶ On parle de *portée lexicale* (ou *statique*) des variables si les variables libres d'une fonction prennent leur valeur dans *l'environnement lexicale*.
- ▶ L'environnement lexicale est celui de la définition.
- ▶ Dans le code il correspond au contexte de la définition de la fonction.

## Exemple : Racket

► Considérons le code Racket suivant :

- `(define x "coucou")`

```
(define (foo)
  (displayln x))
```

```
(foo) ;; affiche "coucou"
```

```
(let ((x "salut"))
  (foo)) ;; affiche "coucou"
```

## Exemple : OCaml

► Considérons le code OCaml suivant :

- `let x = "coucou" ;;`

```
let foo () = print_string x ;;
```

```
foo () ;; (* affiche "coucou" *)
```

```
let x = "salut" in
```

```
  foo () ;; (* affiche "coucou" *)
```

- ▶ Une *clôture* est une fonction accompagnée de son environnement lexical (au moins de ce qui est nécessaire à lier les variables libres de la fonction).
- ▶ Parmi les langages à portée lexicale, certains font leurs clôtures *par références* et d'autres *par valeurs*.

## Clôtures par références

- ▶ Dans certains langages les clôtures capturent une référence à leur environnement lexical.
- ▶ Cela veut dire que si celui-ci est modifié, cela influera sur la clôture.

## Exemple : Racket

► Considérons le code Racket suivant :

- `(define x "coucou")`

```
(define (foo)
  (displayln x))
```

```
(foo) ;; affiche "coucou"
```

```
(set! x "salut")
```

```
(foo) ;; affiche "salut"
```

- Remarque : l'utilisation de `set!` n'est pas idiomatique, la *mutabilité* ne l'étant généralement pas dans la programmation fonctionnelle.



## Clôtures par valeurs

- ▶ Dans certains langages, les clôtures capturent la valeur de leur environnement lexical.
- ▶ Dans ce cas plus de surprise : le résultat d'une fonction ne peut plus dépendre que de ses paramètres.
- ▶ Une fois la clôture créée plus rien ne peut la modifier.

## Clôtures par valeurs

- ▶ Dans certains langages, les clôtures capturent la valeur de leur environnement lexical.
- ▶ Dans ce cas plus de surprise : le résultat d'une fonction ne peut plus dépendre que de ses paramètres.
- ▶ Une fois la clôture créée plus rien ne peut la modifier.
  - Bon, sauf si on utilise des choses pas propre fonctionnellement, comme des références/pointeurs.

## Exemple : OCaml

► Considérons le code OCaml suivant :

- `let x = "coucou" ;;`

```
let foo () = print_string x ;;
```

```
foo () ;; (* affiche "coucou" *)
```

```
let x = "salut" ;;
```

```
foo () ;; (* affiche "coucou" *)
```

- ▶ L'analyse de portée consiste à vérifier que les variables utilisées peuvent bien l'être.
- ▶ Comme on vient de le voir les règles à respecter dépendent des langages.
- ▶ Dans certains cas, elles seront très strictes et empêcheront complètement le plantage à l'exécution (portée lexicale et clôture par valeur).
- ▶ Dans d'autres cas, les règles plus souples permettront moins de vérifications statiques (mais plus de souplesse dans le développement).

- ▶ Pour faire cette analyse, il est nécessaire de maintenir un *environnement*.
- ▶ Il peut simplement s'agir d'une liste des variables accessibles au point du programme en train d'être analysé\*.
- ▶ Cependant attention aux spécificités du langage :
  - il peut être nécessaire de distinguer l'environnement local de l'environnement global.

- ▶ Pour faire cette analyse, il est nécessaire de maintenir un *environnement*.
- ▶ Il peut simplement s'agir d'une liste des variables accessibles au point du programme en train d'être analysé\*.
- ▶ Cependant attention aux spécificités du langage :
  - il peut être nécessaire de distinguer l'environnement local de l'environnement global.

\* Cependant on fait rarement seulement cette analyse, et les suivantes nécessitent de stocker de l'information sur les variables.

- ▶ Le *typage* est l'activité principale de l'analyse sémantique.
- ▶ Il s'agit de vérifier que le programme est correctement typé, par exemple :
  - que les fonctions reçoivent le bon nombre d'arguments,
  - que les types des arguments sont bien ceux attendus par la fonction,
  - que les types des valeurs affectées correspondent à ceux des variables,
  - ...
- ▶ Encore une fois, les propriétés du langage vont fortement influencer ce qu'on va devoir/pouvoir faire.

## Typage dynamique

- ▶ Les langages *dynamiquement typés* attribuent des types aux *valeurs* mais pas aux *variables*.
- ▶ Le type des variables ne peut donc pas toujours être statiquement connu.



# Typage statique

- ▶ Les langages *statiqument typés* attribuent des types aux valeurs et aux *variables*.
- ▶ Cela permet plus de vérifications statiques.

# Typage faible

- ▶ Les langages *faiblement typés* autorisent les valeurs à changer de type automatiquement quand c'est nécessaire et possible.

- ▶ Les langages *fortement typés* n'autorisent pas les valeurs à changer de types.

- ▶ Voyons des exemples de chaque combinaison.

# Typage dynamique faible

- ▶ Commençons par la combinaison qui permet le moins de vérification...

# Typage dynamique faible

- ▶ Commençons par la combinaison qui permet le moins de vérification...
- ▶ Celle qui ne devrait donc surtout pas être utilisée dans les environnements critiques...

# Typage dynamique faible

- ▶ Commençons par la combinaison qui permet le moins de vérification...
- ▶ Celle qui ne devrait donc surtout pas être utilisée dans les environnements critiques...
- ▶ Comme par exemple là où tout se fait aujourd'hui...

# Typage dynamique faible

- ▶ Commençons par la combinaison qui permet le moins de vérification...
- ▶ Celle qui ne devrait donc surtout pas être utilisée dans les environnements critiques...
- ▶ Comme par exemple là où tout se fait aujourd'hui... le navigateur.



# Typage dynamique faible

- ▶ Commençons par la combinaison qui permet le moins de vérification...
- ▶ Celle qui ne devrait donc surtout pas être utilisée dans les environnements critiques...
- ▶ Comme par exemple là où tout se fait aujourd'hui... le navigateur.
- ▶ Hé oui... **JavaScript !** --'

# Typage dynamique faible

- ▶ Commençons par la combinaison qui permet le moins de vérification...
- ▶ Celle qui ne devrait donc surtout pas être utilisée dans les environnements critiques...
- ▶ Comme par exemple là où tout se fait aujourd'hui... le navigateur.
- ▶ Hé oui... **JavaScript!** --'
  - `foo = "coucou";`  
`foo = 42;`  
`console.log("foo vaut " + foo);`

## Typage dynamique fort

## ▶ Python.

- `foo = "coucou"`  
`foo = 42`  
`print("foo vaut " + foo) # erreur à l'exécution`  
`print("foo vaut " + str(foo)) # on doit explicitement demander la conversion`

## Typage statique faible



C.

- `int foo = 42;`  
`foo = "coucou"; // erreur à la compilation`  
`double x = foo; // conversion int vers double automatique`

# Typage statique fort

▶ La combinaison ultime...

# Typage statique fort

- ▶ La combinaison ultime... OCaml :).

# Typage statique fort

▶ La combinaison ultime... OCaml :).

- #

# Typage statique fort

- ▶ La combinaison ultime... OCaml :).
  - `# let foo = 42 ;;`



## Typage statique fort

▶ La combinaison ultime... OCaml :).

- ```
# let foo = 42 ;;  
  val foo : int = 42  
  #
```

## Typage statique fort

▶ La combinaison ultime... OCaml :).

- ```
# let foo = 42 ;;  
val foo : int = 42  
# print_string foo ;;
```

## Typage statique fort

▶ La combinaison ultime... OCaml :).

- ```
# let foo = 42 ;;  
  val foo : int = 42  
  # print_string foo ;;  
    ^^^
```

Error: This expression has type int but an expression was expected of type  
string

#

► La combinaison ultime... OCaml :).

- ```
# let foo = 42 ;;  
  val foo : int = 42  
  # print_string foo ;;  
    ^^^
```

Error: This expression has type int but an expression was expected of type  
string

```
# print_string ((string_of_int 42) ^ "\n") ;;
```

## Typage statique fort

## ▶ La combinaison ultime... OCaml :).

- ```
# let foo = 42 ;;  
val foo : int = 42  
# print_string foo ;;  
      ^^^
```

Error: This expression has type int but an expression was expected of type  
string

```
# print_string ((string_of_int 42) ^ "\n") ;;  
42  
- : unit = ()  
#
```

- ▶ L'*inférence de types* consiste à déduire automatiquement le type d'une expression à partir des types connus (par exemple ceux des valeurs).
- ▶ Le but est alors de trouver le type le plus général possible qui soit compatible avec l'expression.
- ▶ Cette fonctionnalité nécessite généralement que le langage soit statiquement typé.

## Exemple : OCaml

- ▶ L'inférence de type est assez puissante :
  - #

## Exemple : OCaml

▶ L'inférence de type est assez puissante :

- ```
# let rec map f l =  
    match l with  
    | hd :: tl -> (f hd) :: (map f tl)  
    | []       -> []  
;;
```



## Exemple : OCaml

► L'inférence de type est assez puissante :

```
• # let rec map f l =  
    match l with  
    | hd :: tl -> (f hd) :: (map f tl)  
    | []       -> []  
    ;;  
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>  
#
```

## Exemple : OCaml

► L'inférence de type est assez puissante :

```
• # let rec map f l =  
    match l with  
    | hd :: tl -> (f hd) :: (map f tl)  
    | []        -> []  
    ;;  
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>  
# map int_of_string ;;
```

## Exemple : OCaml

► L'inférence de type est assez puissante :

```
• # let rec map f l =  
    match l with  
    | hd :: tl -> (f hd) :: (map f tl)  
    | []       -> []  
    ;;  
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>  
# map int_of_string ;;  
- : string list -> int list = <fun>  
#
```

## Exemple : OCaml

► L'inférence de type est assez puissante :

```
• # let rec map f l =  
    match l with  
    | hd :: tl -> (f hd) :: (map f tl)  
    | []        -> []  
    ;;  
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>  
# map int_of_string ;;  
- : string list -> int list = <fun>  
# map int_of_string [ 1 ; 2 ; 3 ] ;;
```

## Exemple : OCaml

► L'inférence de type est assez puissante :

```

• # let rec map f l =
    match l with
    | hd :: tl -> (f hd) :: (map f tl)
    | []       -> []
    ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map int_of_string ;;
- : string list -> int list = <fun>
# map int_of_string [ 1 ; 2 ; 3 ] ;;
                        ^

```

Error: This expression has type int but an expression was expected of type string

```
#
```

## Exemple : OCaml

► L'inférence de type est assez puissante :

```
• # let rec map f l =  
    match l with  
    | hd :: tl -> (f hd) :: (map f tl)  
    | []        -> []  
    ;;  
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>  
# map int_of_string ;;  
- : string list -> int list = <fun>  
# map int_of_string [ 1 ; 2 ; 3 ] ;;  
                        ^
```

Error: This expression has type int but an expression was expected of type string

```
# map int_of_string [ "1" ; 2 ; 3 ] ;;
```

## Exemple : OCaml

► L'inférence de type est assez puissante :

```
• # let rec map f l =  
    match l with  
    | hd :: tl -> (f hd) :: (map f tl)  
    | []        -> []  
    ;;  
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>  
# map int_of_string ;;  
- : string list -> int list = <fun>  
# map int_of_string [ 1 ; 2 ; 3 ] ;;  
                        ^
```

Error: This expression has type int but an expression was expected of type string

```
# map int_of_string [ "1" ; 2 ; 3 ] ;;  
                        ^
```

Error: This expression has type int but an expression was expected of type string

```
#
```

## Exemple : OCaml

► L'inférence de type est assez puissante :

```
• # let rec map f l =
    match l with
    | hd :: tl -> (f hd) :: (map f tl)
    | []       -> []
  ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map int_of_string ;;
- : string list -> int list = <fun>
# map int_of_string [ 1 ; 2 ; 3 ] ;;
                        ^
```

Error: This expression has type int but an expression was expected of type string

```
# map int_of_string [ "1" ; 2 ; 3 ] ;;
                        ^
```

Error: This expression has type int but an expression was expected of type string

```
# map int_of_string [ "1" ; "2" ; "3" ] ;;
```



## Exemple : OCaml

► L'inférence de type est assez puissante :

```
• # let rec map f l =
    match l with
    | hd :: tl -> (f hd) :: (map f tl)
    | []        -> []
  ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map int_of_string ;;
- : string list -> int list = <fun>
# map int_of_string [ 1 ; 2 ; 3 ] ;;
                        ^
```

Error: This expression has type int but an expression was expected of type string

```
# map int_of_string [ "1" ; 2 ; 3 ] ;;
                        ^
```

Error: This expression has type int but an expression was expected of type string

```
# map int_of_string [ "1" ; "2" ; "3" ] ;;
- : int list = [1; 2; 3]
#
```

- ▶ Le but d'un *système de types* est de limiter le nombre d'expressions sémantiquement valides dans un langage.
- ▶ Cela permet en échange de s'assurer de certaines propriétés à l'exécution.
- ▶ Exemples :
  - empêcher les bug liés au mauvais typage (évidemment).
  - empêcher l'écriture de programme qui ne termine pas (et donc faire perdre la Turing-complétude à son langage).
  - s'assurer qu'une fonction fait bien ce qu'elle est censé faire, si on arrive à encoder ses spécifications dans le système de type.
- ▶ Il existe des systèmes de types très puissants qui permettent de faire de la vérification de programmes (correspondance de Curry-Howard).

- ▶ Comme on vient de le voir il n'y a pas une unique façon de vérifier le typage d'un programme.
- ▶ Il y a cependant quelques méthodes "génériques".

- ▶ Comme pour la vérification de portée, on va devoir utiliser des environnements.
- ▶ Cette fois, ils associent à chaque variables des informations sur son type permettant de vérifier non seulement que la variable existe dans le contexte mais aussi qu'elle est utilisée correctement :
  - variables : type ;
  - fonctions : arité, types attendus en argument, types de retour.

## Arbres d'inférence

- ▶ L'idée générale est de vérifier que les informations de types qu'on possède sont compatibles avec celles qu'on doit deviner et avec le système de types du langage.
- ▶ On va faire cela en construisant un *arbre d'inférence*.
- ▶ Cette arbre correspond en fait à des étiquettes d'information sur les nœuds de notre AST.
- ▶ Il va servir à :
  - deviner le type de certaines expressions faisant appel à des fonctions polymorphes (ajout de contraintes, inférences),
  - produire des erreurs plus ou moins précises,
  - rajouter du code pour la conversion de type dans les langages faiblement typés.

- ▶ Gawi, pour “*gates and wires*” est un petit langage de netlist, qui étend les exemples basé sur les expressions booléennes qu’on a vu jusqu’à présent.
- ▶ En Gawi, on peut définir de nouveaux blocs à partir des portes logiques de base.
- Regardons ensemble quelques petits exemples, puis un front-end complet (analyse lexicale, syntaxique, et sémantique) pour gawi.