

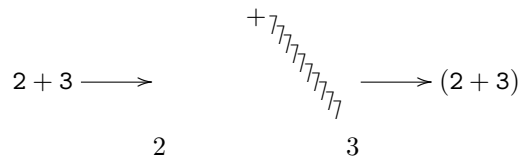
# Compilation — TD 4

J. Boender & G. Henry

2009–2010

Vous savez déjà qu'un *parseur* est un module qui traduit une syntaxe concrète, représentée comme une suite de caractères, en une syntaxe abstraite, représentée par un arbre. Un *pretty-printer* est l'inverse d'un parseur : c'est un module qui, étant donné un arbre de syntaxe abstraite, produit une représentation syntaxique concrète de cet arbre.

Le pretty-printer est un *inverse à droite* du parseur : en partant d'un arbre de syntaxe abstraite, appliquer le pretty-printer puis le parseur nous ramène à l'arbre de départ. Il n'est cependant normalement pas un *inverse à gauche* du parseur : appliquer le parseur à un programme en syntaxe concrète puis le pretty-printer à l'arbre résultant donne un programme certes équivalent mais pas nécessairement identique au programme de départ.



Le but de ce TP est d'écrire un pretty-printer pour le langage CTigre utilisé dans le projet. Cet exercice vous permettra de vous familiariser avec la syntaxe abstraite utilisée ; de plus, vous pourrez vous servir de ce pretty-printer pour votre projet.

## Analyse syntaxique

1. Écrivez un fichier `main.ml` qui effectue l'analyse syntaxique des fichiers passés sur la ligne de commande. Modifiez le `Makefile` fourni pour qu'il compile ce nouveau module. Compilez et testez votre embryon de compilateur avec des programmes CTigre syntaxiquement correct ou incorrects.

## « Pretty-print » basique

2. Dans un module `PrettyPrint`, écrivez une fonction `print` qui a le type `Ast.rawexp → unit` et qui affiche « ... » sur le terminal. Modifiez votre programme principal pour qu'il appelle votre fonction avec les arbres de syntaxe produit par l'analyseur syntaxique.

3. Modifiez la fonction `print` pour qu'elle imprime correctement les entiers, les variables, les chaînes, les expressions séquentielles et les applications. Testez-la sur le programme suivant :

```
print("Je vais afficher 2.");
printint(2)
```

Vérifiez que le résultat affiché par votre fonction `print` peut-être correctement lu par l'analyseur syntaxique ! Faites de même après chacune des modifications que vous apporterez à cette fonction.

4. Étendez votre fonction aux opérandes en paranthésant systématiquement les expressions. Testez le programme résultant sur le programme suivant :

```
print("1 + 2 font ");
printint(1 + 2)
```

5. Étendez ensuite votre programme aux conditionnelles, et testez-le sur un exemple que vous choisirez.

6. Étendez progressivement votre programme aux déclarations de fonction, aux déclaration de variables et aux affectations. Testez-le résultat sur la factorielle récursive donnée dans le projet ainsi que sur un programme impératif élémentaire.

7. Étendez votre pretty-printer à la totalité du langage CTigre.

## « Pretty-print » amélioré

8. Votre pretty-printer est peu économe en parenthèses. Par exemple, il affiche « `2 + 3 + 4` » comme « `((2 + 3) + 4)` » et « `2 + 3 * 5` » comme « `((2 + 3) * 5)` ». Modifiez-le pour qu'il évite d'afficher des parenthèses lorsque ce n'est pas nécessaire. Il faudra bien-sûr prendre en compte les règles d'associativité à gauche ainsi que les priorités relatives des opérateurs CTigre.

## Programme principal

9. Modifiez votre programme principal pour qu'il respecte les options suivantes sur la ligne de commande :

```
-ast      imprime l'AST à l'aide de AstPrint.print_raw (fournie);
-ppast    imprime joliment l'AST à l'aide de votre fonction.
```

## Interface for module Ast

```
open Symbol
open Label

type direction_ ag = Up | Down

type oper =
  | PlusOp | MinusOp | TimesOp | DivideOp
  | AndOp | OrOp
  | EqOp | NeqOp | LtOp | LeOp | GtOp | GeOp

type typename = symbol
type eldname = symbol

type core_type =
  | Typ_alias of typename
  | Typ_array of typename
  | Typ_record of eld list

and eld = { _name : eldname; _typ : typename }

type typedec = symbol × core_type
```

**Parametrized AST** The type of ctigre expressions is parametrized by the type of variable symbols ('v) and function symbols : at call site ('fc) and at definition ('fd)

```
type ('v, 'fc, 'fd) exp =
  | VarExp of ('v, 'fc, 'fd) var
  | NilExp
  | IntExp of int
  | StringExp of strin
  | CharExp of char
  | Apply of 'fc × ('v, 'fc, 'fd) exp list
  | RecordExp of ( eldname × ('v, 'fc, 'fd) exp) list × typename
  | SeqExp of ('v, 'fc, 'fd) exp × ('v, 'fc, 'fd) exp
  | IfExp of ('v, 'fc, 'fd) exp × ('v, 'fc, 'fd) exp × ('v, 'fc, 'fd) exp option
  | WhileExp of ('v, 'fc, 'fd) exp × ('v, 'fc, 'fd) exp
  | ForExp of ('v, 'fc, 'fd) forexp
  | LetVarExp of ('v, 'fc, 'fd) vardec list × ('v, 'fc, 'fd) exp
  | LetFunExp of ('v, 'fc, 'fd) fundec list × ('v, 'fc, 'fd) exp
  | TypeExp of typedec list × ('v, 'fc, 'fd) exp
  | ArrayExp of ('v, 'fc, 'fd) arrayexp
  | Opexp of oper × ('v, 'fc, 'fd) exp × ('v, 'fc, 'fd) exp
  | AssignExp of ('v, 'fc, 'fd) var × ('v, 'fc, 'fd) exp

and ('v, 'fc, 'fd) var =
  | SimpleVar of 'v
  | FieldVar of ('v, 'fc, 'fd) var × eldname
  | SubscriptVar of ('v, 'fc, 'fd) var × ('v, 'fc, 'fd) exp
```

```

and ('v, 'fc, 'fd) forexp =
  { for_var : 'v;
    for_lo : ('v, 'fc, 'fd) exp;
    for_hi : ('v, 'fc, 'fd) exp;
    for_dir : direction_ ag;
    for_body : ('v, 'fc, 'fd) exp }

and ('v, 'fc, 'fd) arrayexp = { a_typ : typename;
                               a_size : ('v, 'fc, 'fd) exp;
                               a_init : ('v, 'fc, 'fd) exp }

and ('v, 'fc, 'fd) vardec = { var_id : 'v;
                             var_typ : typename option;
                             var_init : ('v, 'fc, 'fd) exp }

and ('v, 'fc, 'fd) fundec = { fun_id : 'fd;
                             fun_params : param list;
                             fun_res : typename option;
                             fun_body : ('v, 'fc, 'fd) exp }

and param = { p_name : symbol; p_typ : typename option }

```

**Concrete AST** The two instance of *exp* we will use :

*rawexp* will simply have *symbol* for variables and functions

*attrexpr* will have :

- Temp or level/offset for variables,
- Label and level for function definition,
- Internal or predefined for function call.

type *rawexp* = (*symbol*, *symbol*, *symbol*) *exp*

type *varpos* =  
 | *Stack* of *int* × *int* (\* Allocation in the stack (level, offset) \*)  
 | *Temp* of *Temp.temp* (\* Allocation in a temp \*)

type *attrvar* = { *v\_name* : *symbol*; mutable *v\_pos* : *varpos* }

type *attrfundef* = { *f\_name* : *lbl*; *f\_level* : *int*; }

type *attrfuncall* =  
 | *Internal* of *attrfundef*  
 | *Predefined* of *lbl*

type *attrexpr* = (*attrvar*, *attrfuncall*, *attrfundef*) *exp*