Compilation

Linéarisation du contrôle

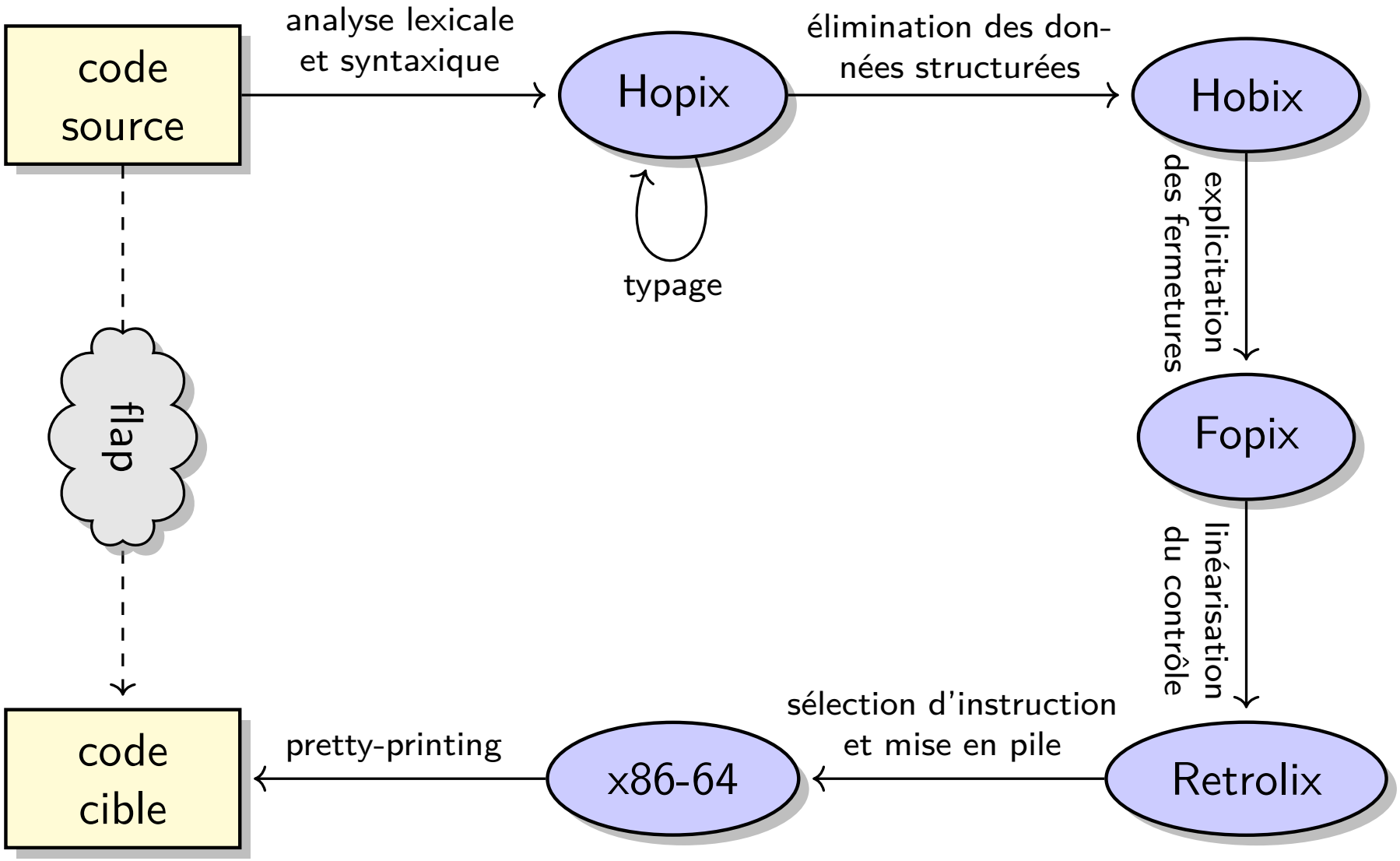
Adrien Guatto

Master 1 Informatique
2022–2023

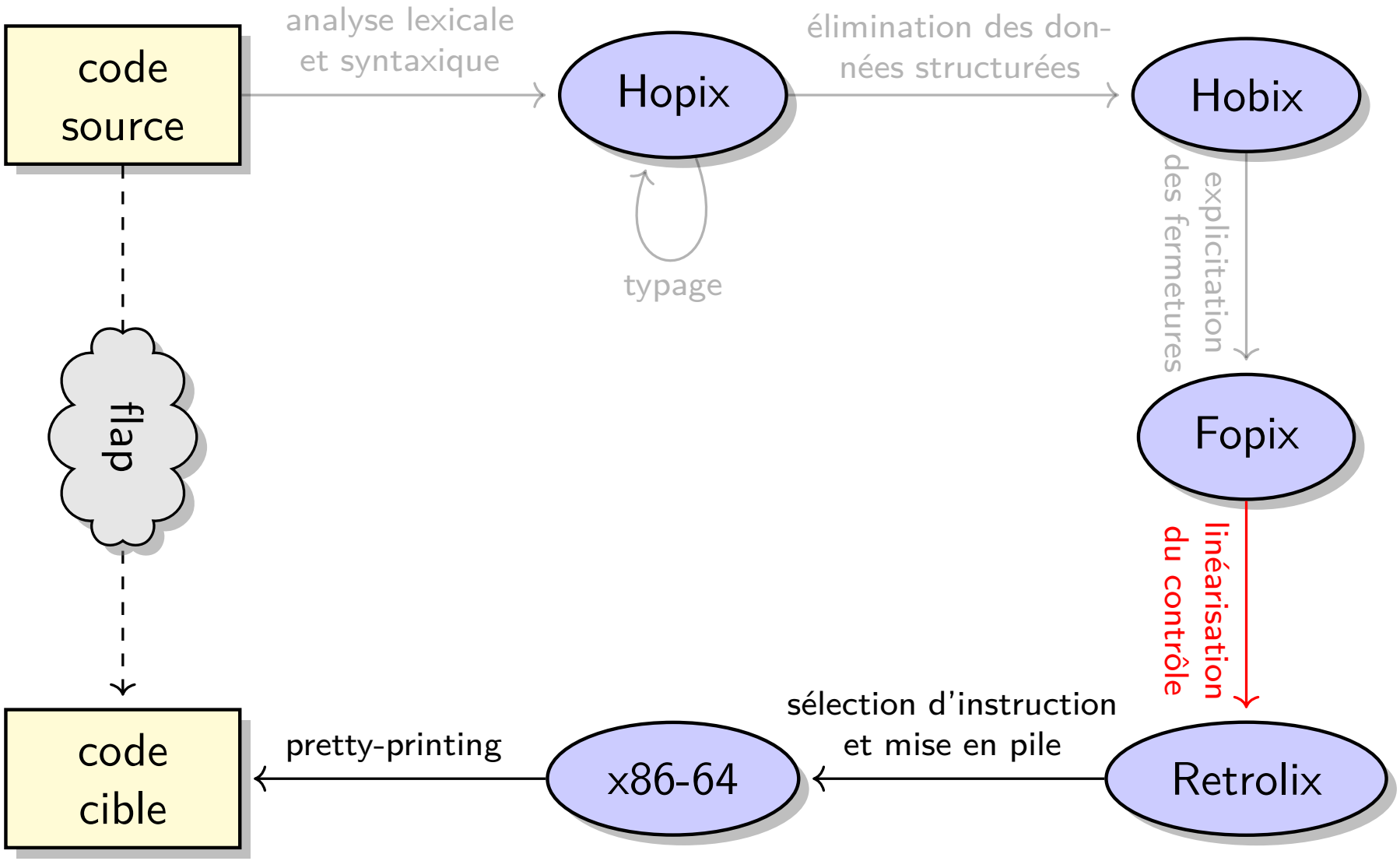
Plan

Plan

Le flot de compilation de flap



Le flot de compilation de flap



Rappels : le langage Fopix

```
type program = definition list

and definition =
| DefineValue      of          identifier * expression
| DefineFunction   of function_identifier * formals * expression
| ExternalFunction of function_identifier * int

and expression =
| Literal of literal
| Variable of identifier
| Define of identifier * expression * expression
| FunCall of function_identifier * expression list
| UnknownFunCall of expression * expression list
| While of expression * expression
| IfThenElse of expression * expression * expression
| Switch of expression * expression option array * expression option

and formals = identifier list
```

Le langage Retrolix (1/2)

```
type program = definition list

and definition =
| DValues    of identifier list * block
| DFunction  of function_identifier * identifier list * block
| DExternalFunction of function_identifier

and block = identifier list * labelled_instruction list

and labelled_instruction = label * instruction
```

Le langage Retrolix (2/2)

```
and instruction =
| Call of rvalue * rvalue list * bool
| Ret
| Assign of lvalue * op * rvalue list
| Jump of label
| ConditionalJump of condition * rvalue list * label * label
| Switch of rvalue * label array * label option
| Comment of string
| Exit

and op = Copy | Add | Mul | Div | Sub | And | Or
and condition = GT | LT | GTE | LTE | EQ

and rvalue = [ lvalue | `Immediate of literal ]

and lvalue = [ `Variable of identifier | `Register of register ]

and register = RId of string
```

De Fopix à Retrolix

Question

Quelles sont les similarités et différences entre Fopix et Retrolix ?

De Fopix à Retrolix

Question

Quelles sont les similarités et différences entre Fopix et Retrolix ?

- Fopix et Retrolix partagent leur notion de littéraux.

De Fopix à Retrolix

Question

Quelles sont les similarités et différences entre Fopix et Retrolix ?

- Fopix et Retrolix partagent leur notion de littéraux.
- Retrolix est un langage à base d'instructions atomiques élémentaires.

De Fopix à Retrolix

Question

Quelles sont les similarités et différences entre Fopix et Retrolix ?

- Fopix et Retrolix partagent leur notion de littéraux.
- Retrolix est un langage à base d'instructions atomiques élémentaires.
  - Fopix et ses prédécesseurs ont des expressions hiérarchiques.

De Fopix à Retrolix

Question

Quelles sont les similarités et différences entre Fopix et Retrolix ?

- Fopix et Retrolix partagent leur notion de littéraux.
- Retrolix est un langage à base d'instructions atomiques élémentaires.
  - Fopix et ses prédécesseurs ont des expressions hiérarchiques.
- Son flot de contrôle est contrôlé par des sauts plutôt que structuré.

De Fopix à Retrolix

Question

Quelles sont les similarités et différences entre Fopix et Retrolix ?

- Fopix et Retrolix partagent leur notion de littéraux.
- Retrolix est un langage à base d'instructions atomiques élémentaires.
  - Fopix et ses prédécesseurs ont des expressions hiérarchiques.
- Son flot de contrôle est contrôlé par des sauts plutôt que structuré.
  - Toute instruction a une étiquette qui peut servir de cible aux sauts.

De Fopix à Retrolix

Question

Quelles sont les similarités et différences entre Fopix et Retrolix ?

- Fopix et Retrolix partagent leur notion de littéraux.
- Retrolix est un langage à base d'instructions atomiques élémentaires.
  - Fopix et ses prédécesseurs ont des expressions hiérarchiques.
- Son flot de contrôle est contrôlé par des sauts plutôt que structuré.
  - Toute instruction a une étiquette qui peut servir de cible aux sauts.
- Certains appels sont marqués *terminaux* via un drapeau booléen.

De Fopix à Retrolix

Question

Quelles sont les similarités et différences entre Fopix et Retrolix ?

- Fopix et Retrolix partagent leur notion de littéraux.
- Retrolix est un langage à base d'instructions atomiques élémentaires.
  - Fopix et ses prédécesseurs ont des expressions hiérarchiques.
- Son flot de contrôle est contrôlé par des sauts plutôt que structuré.
  - Toute instruction a une étiquette qui peut servir de cible aux sauts.
- Certains appels sont marqués *terminaux* via un drapeau booléen.
- Ses variables sont locales à un bloc, et il n'y a pas de blocs imbriqués.

De Fopix à Retrolix

Question

Quelles sont les similarités et différences entre Fopix et Retrolix ?

- Fopix et Retrolix partagent leur notion de littéraux.
- Retrolix est un langage à base d'instructions atomiques élémentaires.
  - Fopix et ses prédécesseurs ont des expressions hiérarchiques.
- Son flot de contrôle est contrôlé par des sauts plutôt que structuré.
  - Toute instruction a une étiquette qui peut servir de cible aux sauts.
- Certains appels sont marqués *terminaux* via un drapeau booléen.
- Ses variables sont locales à un bloc, et il n'y a pas de blocs imbriqués.
- En plus des variables, Retrolix dispose d'une notion de *registre*.

De Fopix à Retrolix

Question

Quelles sont les similarités et différences entre Fopix et Retrolix ?

- Fopix et Retrolix partagent leur notion de littéraux.
- Retrolix est un langage à base d'instructions atomiques élémentaires.
  - Fopix et ses prédécesseurs ont des expressions hiérarchiques.
- Son flot de contrôle est contrôlé par des sauts plutôt que structuré.
  - Toute instruction a une étiquette qui peut servir de cible aux sauts.
- Certains appels sont marqués *terminaux* via un drapeau booléen.
- Ses variables sont locales à un bloc, et il n'y a pas de blocs imbriqués.
- En plus des variables, Retrolix dispose d'une notion de *registre*.
- Il suit la *convention d'appel* de la machine.

Observation

Retrolix est un langage de bas-niveau proche de celui de la machine.

Le langage machine

- Votre processeur implémente un langage de programmation.
  - Appelé *langage machine* ou *d'assemblage*, improprement *assembleur*.
  - Généralement commun à une famille de processeurs ; évolue lentement.
  - Le plus petit dénominateur commun du logiciel que vous exécutez.
- Celui-ci est explicite et souvent baroque, voire très baroque.
  - Plus ou moins grand nombre d'instructions plus ou moins orthogonales.
  - Des effets de bords omniprésents, e.g., les *codes de condition*.
- C'est la fondation des couches basses du système (noyau, libc…).
  - Elles agissent de concert pour définir et respecter une *convention d'appel* qui permet l'interopérabilité entre procédures.

Le langage machine

- Votre processeur implémente un langage de programmation.
  - Appelé *langage machine* ou *d'assemblage*, improprement *assembleur*.
  - Généralement commun à une famille de processeurs ; évolue lentement.
  - Le plus petit dénominateur commun du logiciel que vous exécutez.
- Celui-ci est explicite et souvent baroque, voire très baroque.
  - Plus ou moins grand nombre d'instructions plus ou moins orthogonales.
  - Des effets de bords omniprésents, e.g., les *codes de condition*.
- C'est la fondation des couches basses du système (noyau, libc…).
  - Elles agissent de concert pour définir et respecter une *convention d'appel* qui permet l'interopérabilité entre procédures.

Très bientôt

Le langage machine des processeurs x86-64 d'Intel et AMD.

Le module `FopixToRetrolix`

Résumé de la tâche de la passe

- Passer d'un langage d'expressions structurées à un langage "à plat".
- Implémenter la convention d'appel de la machine cible.

Le module `FopixToRetrolix`

Résumé de la tâche de la passe

- Passer d'un langage d'expressions structurées à un langage "à plat".
- Implémenter la convention d'appel de la machine cible.

La linéarisation du contrôle

- Aplatir les expressions en générant étiquettes et variables.
- Le code généré est très gros mais facile à produire.

Plan

Traduire les expressions arithmétiques

Question

Comment traduire la définition `let r = 1 - (3 * 4)` ?

Traduire les expressions arithmétiques

Question

Comment traduire la définition `let r = 1 - (3 * 4)` ?

```
globals (r)
local x1, x2, x3, x4:
  x1 <- copy 1;
  x2 <- copy 3;
  x3 <- copy 4;
  x4 <- mul x2, x3;
  r  <- add x1, x4;
  ret;
end
```

(Les étiquettes, ici superflues, ont été omises.)

Traduire les expressions arithmétiques

Question

Comment traduire la définition `let r = 1 - (3 * 4)` ?

```
globals (r)
local x1, x2, x3, x4:
  x1 <- copy 1;
  x2 <- copy 3;
  x3 <- copy 4;
  x4 <- mul x2, x3;
  r  <- add x1, x4;
  ret;
end
```

(Les étiquettes, ici superflues, ont été omises.)

Remarque

On peut facilement générer moins de variables locales.

Traduire les conditionnelles

Question

Comment traduire la définition `let r = if x = 0 then 0 else y / x` ?

Traduire les conditionnelles

Question

Comment traduire la définition `let r = if x = 0 then 0 else y / x` ?

```
globals (r)
locals x1, x2, x3, x4, x5:
    x1 <- copy 1;
    x2 <- copy x;
    jumpif eq x1, x2 -> lE, lT
lT: r <- copy 0;
    jmp lK
lE: x3 <- y;
    x4 <- x;
    r <- div x3, x4;
lK: ret;
end
```

Traduire le flot de contrôle

Question

Comment traduire la définition `while(x[0]>=0){x[0]:=x[0]-1}; x[0]` ?

Traduire le flot de contrôle

Question

Comment traduire la définition `while(x[0]>=0){x[0]:=x[0]-1}; x[0]` ?

```
globals (r)
locals x1, x2, x3, x4, x5:
lT: x1 <- read_block(x, 0);
    x2 <- copy 0;
    jumpif gte x1, x2 -> lK, lB
lB: x3 <- read_block(x, 0);
    x4 <- sub x3, 1;
    write_block(x, 0, x4);
    jump lT
lK: r <- read_block(x, 0);
    ret;
end
```

Une traduction simplifiée (1/2)

```
(* Micro Fopix *)
type exp = Id of id | Lit of int
         | BinOp of binop * exp * exp
         | IfZero of exp * exp * exp
         | WhileNotZero of exp * exp
         | Let of id * exp * exp
(* Micro Retrolix *)
type label = string
type lvalue = [ `Var of id ]
type rvalue = [ lvalue | `Imm of int ]
type op = Bin of binop | Copy
type insn = Assign of lvalue * op * rvalue list
          | ConditionalJumpZero of rvalue * label * label
          | Jump of label
          | Comment of string
          | Call of string * rvalue list * bool
type labelled_insn = label * insn
type block = id list * labelled_insn list
```

Une traduction simplifiée (2/3)

```
let rec translate (r : lvalue) (e : exp) : block = match e with
| Id x -> [], [copy ~dst:r ~src:(`Var x)]
| Lit n -> [], [copy ~dst:r ~src:(`Imm n)]
| BinOp (o, e1, e2) ->
  let rv1, (loc1, insn1) = rvalue_of_exp e1 in
  let rv2, (loc2, insn2) = rvalue_of_exp e2 in
  loc1 @ loc2, insn1 @ insn2 @ [label (Assign (r, Bin o, [rv1; rv2]))]
| IfZero (e_cond, e_t, e_f) ->
  let rv_cond, (loc_cond, insn_cond) = rvalue_of_exp e_cond in
  let lt, (loc_t, insn_t) = locate_translate r e_t in
  let lf, (loc_f, insn_f) = locate_translate r e_f in
  let lk = fresh_label () in
  loc_cond @ loc_t @ loc_f,
  insn_cond @ [label (ConditionalJumpZero (rv_cond, lt, lf))]
  @ insn_t @ insn_f @ [(lk, Comment "Jonction")]
...
```

Une traduction simplifiée (3/3)

```
let rec translate (r : lvalue) (e : exp) : block = match e with
...
| WhileNotZero (e1, e2) ->
  let x = fresh_id () and lk = fresh_label () in
  let ll, (loc1, insn1) = locate_translate (`Var x) e1 in
  let lb, (loc2, insn2) = locate_translate r e2 in
  x :: loc1 @ loc2,
  insn1 @ [label (ConditionalJumpZero (`Var x, lb, lk))]
  @ insn2 @ [label (Jump ll); (lk, Comment "loop exit")]
| Let (x, e1, e2) ->
  let loc1, insn1 = translate (`Var x) e1 in
  let loc2, insn2 = translate r e2 in
  loc1 @ loc2, insn1 @ insn2
```

Plan

Les conventions d'appel

Qu'est-ce qu'une convention d'appel ?

- Un protocole qui décrit le déroulement des appels au niveau machine.
  1. Gestion de l'adresse de retour (duo appel/retour).
  2. Passage des arguments.
  3. Transmission de la valeur de retour.
  4. Mécanisme de gestion et préservation des données locales.
- Il existe des conventions d'appel privées, spécifiques à un langage.
  - E.g., `ocamlopt` a sa convention spécialisée pour le code fonctionnel.
- Celle du système assure l'interopérabilité entre acteurs.
  - E.g., entre GCC, Clang, la libc ou le noyau Linux.

Les conventions d'appel

Qu'est-ce qu'une convention d'appel ?

- Un protocole qui décrit le déroulement des appels au niveau machine.
  1. Gestion de l'adresse de retour (duo appel/retour).
  2. Passage des arguments.
  3. Transmission de la valeur de retour.
  4. Mécanisme de gestion et préservation des données locales.
- Il existe des conventions d'appel privées, spécifiques à un langage.
  - E.g., `ocamlopt` a sa convention spécialisée pour le code fonctionnel.
- Celle du système assure l'interopérabilité entre acteurs.
  - E.g., entre GCC, Clang, la libc ou le noyau Linux.

Notre convention d'appel : POSIX SystemV x86-64

Standard système *de facto* pour Linux et macOS sur Intel/AMD 64 bits.

La convention d'appel POSIX SystemV x86-64

- Très riche et détaillée, mais Flap n'en utilisera qu'une petite fraction.
- Pour **`FopixToRetrolix`** : passage des arguments et valeur de retour.
  - Les six premiers arguments sont dans des *registres* fixés.
  - Le résultat est passé par un *registre* fixé.
  - (Un *registre* est une petite zone de données modifiable du processeur.)

La convention d'appel POSIX SystemV x86-64

- Très riche et détaillée, mais Flap n'en utilisera qu'une petite fraction.
- Pour **`FopixToRetrolix`** : passage des arguments et valeur de retour.
  - Les six premiers arguments sont dans des *registres* fixés.
  - Le résultat est passé par un *registre* fixé.
  - (Un *registre* est une petite zone de données modifiable du processeur.)

À la découverte de noms cabalistiques (plus de détails bientôt !)

- Les arguments sont passés dans `RDI`, `RSI`, `RDX`, `RCX`, `R8` et `R9`.
- Le résultat est passé dans `RAX`.

La convention d'appel POSIX SystemV x86-64

- Très riche et détaillée, mais Flap n'en utilisera qu'une petite fraction.
- Pour **FopixToRetrolix** : passage des arguments et valeur de retour.
  - Les six premiers arguments sont dans des *registres* fixés.
  - Le résultat est passé par un *registre* fixé.
  - (Un *registre* est une petite zone de données modifiable du processeur.)

À la découverte de noms cabalistiques (plus de détails bientôt !)

- Les arguments sont passés dans `RDI`, `RSI`, `RDX`, `RCX`, `R8` et `R9`.
- Le résultat est passé dans `RAX`.

On traduit donc `let r = f x1 ... x6 x7 ... xN` en

```
%RDI <- copy x1; %RSI <- copy x2; %RDX <- copy x3;
%RCX <- copy x4; %R8 <- copy x5; %R9 <- copy X6;
call f (x7, ..., xN);
r <- copy %RAX;
```

Plan

Rappels au sujet de la récursion terminale

```
let rec fact n = if n <= 0 then 1 else n * fact (n - 1)

let fact' n =
  let rec aux acc n =
    if n <= 0 then acc else aux (n * acc) (n - 1)
  in aux 1 n
```

- L'appel `aux (n * acc) (n - 1)` est terminal, au sens où l'on applique aucune opération au résultat. Contrastez avec `n * fact (n - 1)` !

Rappels au sujet de la récursion terminale

```
let rec fact n = if n <= 0 then 1 else n * fact (n - 1)

let fact' n =
  let rec aux acc n =
    if n <= 0 then acc else aux (n * acc) (n - 1)
  in aux 1 n
```

- L'appel `aux (n * acc) (n - 1)` est terminal, au sens où l'on applique aucune opération au résultat. Contrastez avec `n * fact (n - 1)` !
- La fonction `aux` est récursive terminale parce que tous ses appels récursifs sont terminaux.

Rappels au sujet de la récursion terminale

```
let rec fact n = if n <= 0 then 1 else n * fact (n - 1)

let fact' n =
  let rec aux acc n =
    if n <= 0 then acc else aux (n * acc) (n - 1)
  in aux 1 n
```

- L'appel `aux (n * acc) (n - 1)` est terminal, au sens où l'on applique aucune opération au résultat. Contrastez avec `n * fact (n - 1)` !
- La fonction `aux` est récursive terminale parce que tous ses appels récursifs sont terminaux.
- Les appels terminaux peuvent et doivent être implémentés plus efficacement dans **`RetrolixToX86_64`**. Il faut déjà les détecter !

Calculer les appels terminaux

```
(* Micro Fopix avec appels *)
type exp = Id of id | Lit of int
         | BinOp of binop * exp * exp
         | IfZero of exp * exp * exp
         | WhileNotZero of exp * exp
         | Call of string * exp list
         | Let of id * exp * exp
```

Question

Comment écrire une fonction `tailcalled : exp -> string list` qui calcule les fonctions appelées en position terminale dans son argument ?

Calculer les appels terminaux

```
(* Micro Fopix avec appels *)
type exp = Id of id | Lit of int
         | BinOp of binop * exp * exp
         | IfZero of exp * exp * exp
         | WhileNotZero of exp * exp
         | Call of string * exp list
         | Let of id * exp * exp
```

Question

Comment écrire une fonction `tailcalled : exp -> string list` qui calcule les fonctions appelées en position terminale dans son argument ?

```
let rec tailcalled = function
  | Id _ | Lit _ | BinOp _ | WhileNotZero _ -> []
  | IfZero (_, e1, e2) -> tailcalled e1 @ tailcalled e2
  | Call (f, _) -> [f]
  | Let (_, _, e) -> tailcalled e
```

Calculer les appels terminaux

```
(* Micro Fopix avec appels *)
type exp = Id of id | Lit of int
         | BinOp of binop * exp * exp
         | IfZero of exp * exp * exp
         | WhileNotZero of exp * exp
         | Call of string * exp list
         | Let of id * exp * exp
```

Question

Comment écrire une fonction `tailcalled : exp -> string list` qui calcule les fonctions appelées en position terminale dans son argument ?

```
let rec tailcalled = function
  | Id _ | Lit _ | BinOp _ | WhileNotZero _ -> []
  | IfZero (_, e1, e2) -> tailcalled e1 @ tailcalled e2
  | Call (f, _) -> [f]
  | Let (_, _, e) -> tailcalled e
```

Pour compiler, il faut maintenir l'information de position dans `translate`.

Plan

Conclusion

La linéarisation du contrôle

- Présente sous une forme ou une autre dès lors que le langage source contient des expressions et la cible est à plat.
- Très simple dans la version de base qu'on a présenté.

On va sauter le jalon correspondant.