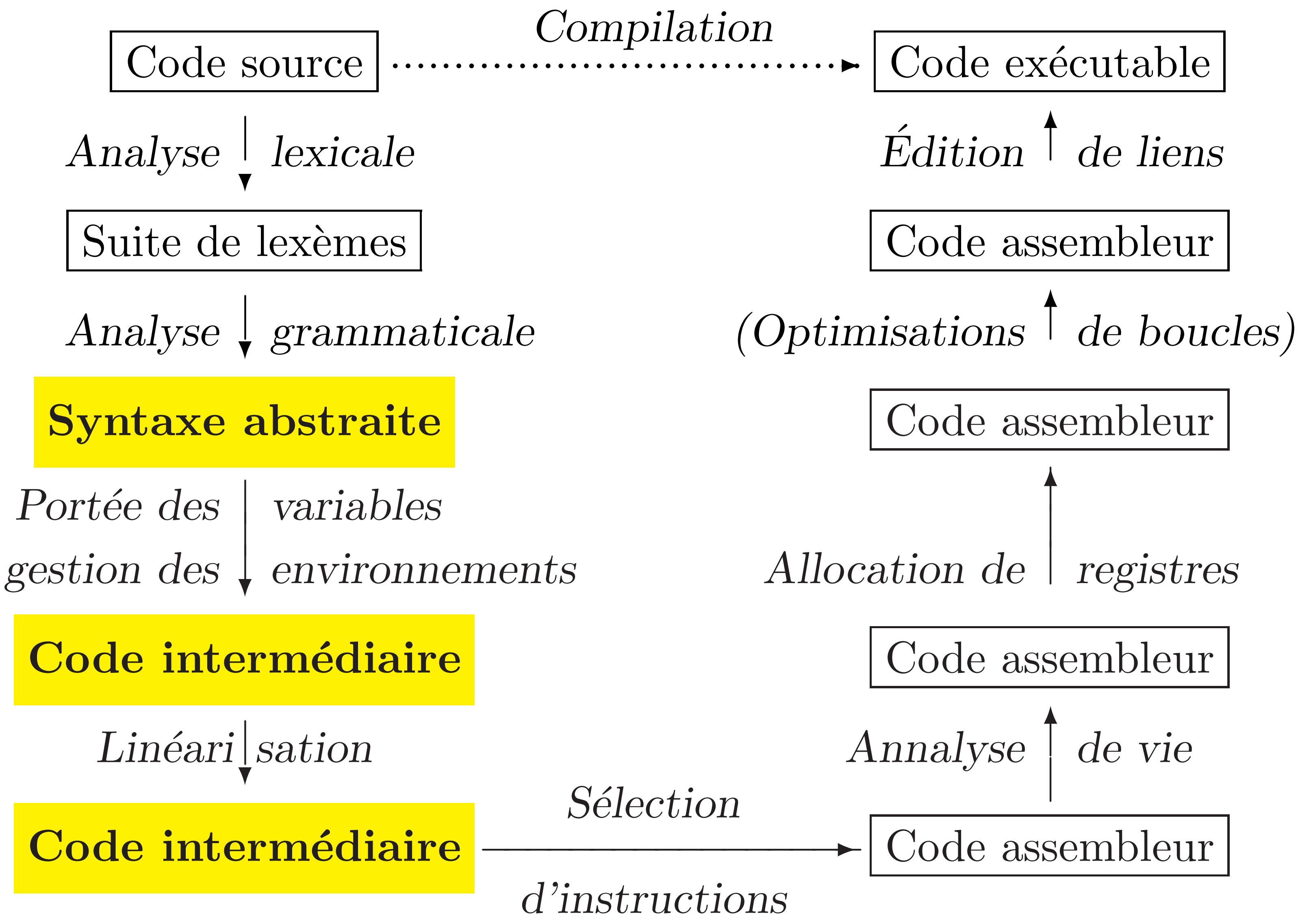
Analyse sémantique (un peu)
Code intermédiaire

Luc Maranget
`http://pauillac.inria.fr/~maranget`
2005


Compilation
Code source
Code exécutable
Analyse lexicale
Édition de liens
Suite de lexèmes
Code assembleur
Analyse grammaticale
(Optimisations de boucles)
Syntaxe abstraite
Code assembleur
Portée des variables
gestion des environnements
Allocation de registres
Code intermédiaire
Code assembleur
Linéarisation
Annalyse de vie
Sélection d'instructions
Code intermédiaire
Code assembleur

Les phases « sémantiques »

Elle reposent sur la sémantique du langage et s'effectue sur l'arbre de syntaxe abstraite.

- Une seule phase réellement indispensable : la génération de code.
- En pratique, bien d'autres phases :
  - Vérification (synthèse) des types.
  - Emploi des noms (pas de doublon des fonctions en Pseudo-Pascal), incluses dans le typage le plus souvent.
  - Optimisations de « haut-niveau ».

Toutes ces phases ont en commun de suivre la sémantique des noms.

Portée lexicale

```
let x = "coucou" in
x ^ (let x = 1 in string_of_int (x+1)) ^ x)
```

Conséquence à l'évaluation :

- créer une liason entre `x` et `"coucou"`, pour évaluer `x ^`,
- créer une liaison entre `x` et `1` pour évaluer `string_of_int (x+1)`,
- retrouver la première liaison pour évaluer `^ x`.

Ou encore :

- `string_of_int (x+1)` est évalué dans un environnement où `x` vaut `1`.
- `x ^` et `^ x` sont évalués dans un environnement où `x` vaut `"coucou"`.

Tables d'associations impératives

Comme les tableaux, les indices sont quelconques.

Nous avons besoin :

- `get : string -> 'a`. Pour accéder aux valeurs des variables.
- `set : string -> 'a -> 'a option`. Pour mettre à jour la valeur d'une variable (renvoie l'ancienne valeur).
- `restore : string -> 'a option -> unit`. Pour remettre une ancienne valeur.

```
let eval = function
| Var x            -> get x
| Let (x, ex, e) ->
    let vx = eval ex in
    let old_vx = set x vx in
    let ve = eval e in
    restore x old_vx ; ve
```

Réalisation, par les tables de hachage bien sûr

```
exception Free of string

let env = Hashtbl create 17

let get x = try Hashtbl.find env x with Not_found -> raise (Free x)

let set x v =
  let old_v = try Some (get x) with Free _ -> None in
  Hashtbl.replace env x v ;
  old_v

let restore x old_v = match old_v with
| None   -> Hashtbl.remove env x      (* détruire la liaison de x *)
| Some v -> Hashtbl.replace env x v  (* restaurer la liaison de x *)
```

Associations fonctionelles

Le style impératif est peu sûr (on a vite fait d'oublier un `restore`).

On souhaite adopter le style de la sémantique avec un environnement passé en argument.

```
let rec eval env = function
| Var x            -> get env x
| Let (x, ex, e) ->
    let vx = eval env ex in
    eval (extend x vx env) e
| ...
```

Plutôt que des tables d'associations, on veut des associations tout court.

Réalisation naïve

Avec des listes bien sûr.

```
(* On pourrait utiliser List.assoc qui fait la même chose *)
let rec get x env = match env with
| []          -> raise (Free x)
| (y,v)::rem ->
   if x=y then v
   else get x rem

let extend x v env = (x,v)::env
```

Noter que les liaisons sont cachées et non plus détruites.

`get` devient innefficace pour les environnements sont un peu grands.

Réalisation efficace

```
(* Module des chaînes ordonnées *)
module OrderedString = struct
  type t = string
  let compare s1 s2 = Pervasive.compare s1 s2 (* ordre standard *)
end

(* Application du foncteur => associations aux chaînes *)
module StringMap = Map.Make OrderedString

let get x env =
  try StringMap.find x env
  with Not_found -> raise (Free x)

let extend x v env = StringMap.add x v env
```

Catégories (ou espaces) de noms

Diverses entitées sont nommées (presque) indépendamment.

- Pseudo-Pascal : fonctions et variables.
- Java : classes, méthodes, variables.
- C : fonctions, variables, étiquettes d'enregistrements etc. (toutefois, interdiction de mélanger fonctions et variables de même nom).

À l'usage, c'est le contexte (syntaxique) qui détermine les espaces de noms :

$$\texttt{f()} \not\sim \texttt{f := ...}$$

Réalisation par des enregistrements, une association par catégorie.

Interface des environnements en Pseudo-Pascal

```
type ('a, 'b) environment

exception VarNotFound of string

val find_var : ('a,'b) environment -> string -> 'a
val find_definition : ('a,'b) environment -> string -> 'b

val create_global : (string * 'a) list -> (string * 'b) list ->
  ('a,'b) environment

val add_local_vars : ('a,'b) environment -> (string * 'a) list ->
  ('a,'b) environment
val change_local_vars : ('a,'b) environment -> (string * 'a) list ->
  ('a,'b) environment
```

Environnements lors de l'exécution

Les noms des variables n'existent plus. Les variables sont des cases :

- Des cases mémoire.
- Des registres machine.

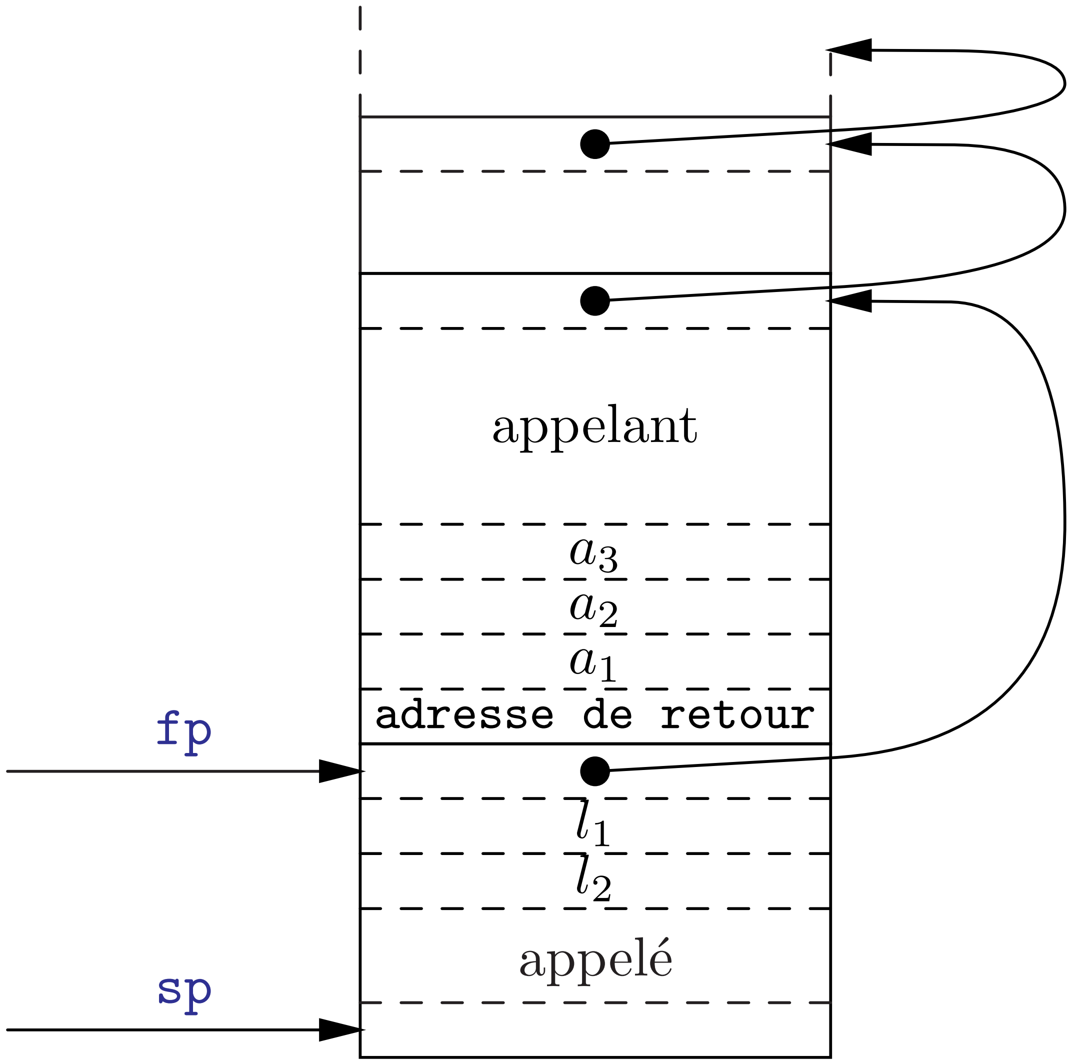
Dans le cas traditionnel :

- Une fonction : l'adresse de son code (étiquette quand même).
- Une variables globale : une case dans le segment statique de données.
- Une variable locale : une case dans la pile.
- Un paramètre : une case dans la pile.

L'allocation en pile provient de la récursivité : les variables (locales) appartiennent aux appels.

Chaque appel possède une tranche (un *frame*) de pile. Il accède aux variables par le fond (`fp`), et dispose du sommet à sa guise (`sp`).

Organisation traditionelle de la pile



Séquence d'appel traditionelle

1. L'appelant empile les paramètres effectifs (et une place pour ranger la valeur de retour).
2. L'appelant exécute une instruction d'appel de sous-routine.
3. L'appelé empile le registre `fp`, puis copie `sp` dans `fp`.
4. L'appelé alloue l'espace nécessaire aux variables locales.
5. L'appelé s'exécute. il doit rendre `sp` comme en 4.
6. L'appelé rend l'espace de pile des variables locales.
7. Il dépile dans `fp`, qui retrouve sa valeur de l'étape 1.
8. Instruction de retour (pop and branch).
9. L'appelant dépile les arguments par lui empilés au début.

Méthode moderne

Il y a bien un frame par appel. Mais...

- On peut se passer de `fp`, si
  - Pas d'allocation arbitraire en pile (`alloca`).
  - Compilation un peu plus complexe acceptée.
  - Pas d'interaction avec un debugger.
- Surtout : les paramètres et variables locales peuvent être en registres (fixés au passage d'arguments pour les premiers).

Conséquence pour la compilation

On ne choisit pas tout de suite si les variables locales vont en registre ou en pile.

Pour le moment, les variables locales sont des « *temporaires* ». C'est un premier exemple d'une construction de code dit « *intermédiaire* ».

Pourquoi un code intermédiaire ?

Partage du travail

- Entre plusieurs langages, et plusieurs machines

$$\left.\begin{array}{r}\text{Java}\\ \text{C}\\ \dots\\ \text{Caml}\\ \text{Ocaml}\end{array}\right\} \boxed{\text{CI}} \left\{\begin{array}{l}\text{MIPS}\\ \text{xx86}\\ \dots\\ \text{Sparc}\\ \text{Byte-code}\end{array}\right.$$

- Entre les différentes constructions d'une même langage ou d'une même machine
  - La syntaxe abstraite comporte trop de constructions voisines
  - Le code assembleur comporte trop d'instructions voisines

Le principe du code intermédiaire

Le code intermédiaire est celui d'une machine idéale.

Les détails dépendant de l'architecture sont relégués à une phase ultérieure de sélection d'instructions.

Quelques caractéristiques du code intermédiaire :

- Les branchements sont explicites.
- Code arborescent (expressions) ou linéaire (instructions)
- Utilise une infinité de registres (temporaires), dont l'utilisation est privilégiée (réversible) et le coût négligé.
- L'adressage en mémoire est une forme séparée qui n'est retenue que lorsque c'est indispensable (irréversible).
- L'appel de fonction reste dans le flou, et sera résolu dans une phase ultérieure. (il depend lourdement du processeur ciblé)

Notre code intermédiaire

Expressions (en arbre)

```
type exp =
    Const of int                          (* Entiers et Booléens *)
  | Name of Gen.label                     (* Adresse mémoire nommée *)
  | Temp of Gen.temp                      (* Lecture d'un temporaire *)
  | Mem of exp                            (* Lecture mémoire *)
  | Bin of binop * exp * exp              (* Opération binaire *)
  | Call of Frame.frame * exp list (* Appel de fonction ou appel système *)
```

Note Données dans le code intermédiare

- Les temporaires Gen.temp
- Les étiquettes Gen.label
- Les fonctions Frame.frame

Instructions (à plat ou presque)

```
and stm =
  | Label of Gen.label                (* Étiquette (dans le code) *)
  | Move_temp of Gen.temp * exp       (* Écriture dans un temporaire *)
  | Move_mem of exp * exp             (* Écriture en mémoire *)
  | Seq of stm list                   (* Séquence d'instructions *)
  | Exp of exp                        (* Expression évaluée pour son effet *)
  | Jump of Gen.label                 (* Saut non conditionnel *)
  | Cjump of                          (* Saut conditionnel  *)
      relop * exp * exp *             (* – Comparaison      *)
        Gen.label * Gen.label         (*  – si vrai/si faux *)
```

Noter les modules de « service » (`Gen` et `Frame`).

Compiler vers le code intermédiaire

Une phase en trois passes

$$\mathrm{SA}\ \underbrace{\xrightarrow{\substack{\text{Générer}\\ \text{du code}}} \ \mathrm{CI}\ \xrightarrow{\substack{\text{Canoniser}\\ \text{Linéariser}}}\ \mathrm{CI}\ \xrightarrow{\substack{\text{Optimiser}\\ \text{le contrôle}}}}_{\text{Compilation}}\ \mathrm{CI}$$

Principe :

- La première passe (simple) introduit des innefficacités, ou fait le travail à moitié.
- Les suivantes (générales) corrigent.

Avantages : Robustesses, modularité, généralité.

Les temporaires

« Sémantique »

- Les temporaires ne sont pas tout à fait des registres, car…
- Les appels de fonction ne modifient pas leur contenu.
- Comme : les variables locales des fonctions récursives.
- Il revient aux phases suivantes de réaliser leur sémantique, avec les moyens de la vraie machine.

Interface (module Gen)

```
type temp
val new_temp : unit -> Gen.temp
```

Génération de code, cas simples

Fonctions $[\![\_]\!]^e_\rho$ (expressions) et $[\![\_]\!]^s_\rho$ (instructions).

- Cas de bases (simples).

$$[\![\texttt{Int}\ n]\!]^e_\rho = \texttt{Const}\ n \quad [\![\texttt{Bool}\ true]\!]^e_\rho = \texttt{Const}\ 1 \quad [\![\texttt{Bool}\ false]\!]^e_\rho = \texttt{Const}\ 0$$

- Étapes inductives (simples).

$$\begin{aligned}
[\![\texttt{Bin}\,(op, e_1, e_2)]\!]^e_\rho &= \texttt{Bin}\,(op, [\![e_1]\!]^e_\rho, [\![e_2]\!]^e_\rho) \\
[\![\texttt{Sequence}\,(s_1; \ldots; s_n)]\!]^s_\rho &= \texttt{Seq}\,[[\![s_1]\!]^s_\rho; \ldots; [\![s_n]\!]^s_\rho]
\end{aligned}$$

- Tableaux (pourrait être bien plus compliqué).

$$\begin{aligned}
[\![\texttt{Geti}\,(e_1, e_2)]\!]^e_\rho &= \\
&\texttt{Mem}\,(\texttt{Bin}\,(\texttt{Plus}\,, [\![e_1]\!]^e_\rho, \texttt{Bin}\,(\texttt{Times}\,, \texttt{Const}\ w, [\![e_2]\!]^e_\rho))) \\
[\![\texttt{Seti}\,(e_1, e_2, e_3)]\!]^s_\rho &= \\
&\texttt{Move_mem}\,(\texttt{Bin}\,(\texttt{Plus}\,, [\![e_1]\!]^e_\rho, \texttt{Bin}\,(\texttt{Times}\,, \texttt{Const}\ w, [\![e_2]\!]^e_\rho)), [\![e_3]\!]^e_\rho)
\end{aligned}$$

Conditionnelle

Test direct ($<$, $\leq$, etc.)

$$[\![\texttt{If}\,(\texttt{Bin}\,(relop, e_1, e_2), s_t, s_f)]\!]^s_\rho = \quad \texttt{Seq}\left[\begin{array}{l} \texttt{Cjump}\,(relop, [\![e_1]\!]^e_\rho, [\![e_2]\!]^e_\rho, l_t, l_f); \\ \texttt{Label}\ l_t; [\![s_t]\!]^s_\rho; \texttt{Jump}\ fi; \\ \texttt{Label}\ l_f; [\![s_f]\!]^s_\rho; \texttt{Jump}\ fi; \\ \texttt{Label}\ fi; \end{array}\right]$$

Sinon, à la C :

$$[\![\texttt{If}\,(e_1, s_t, s_f)]\!]^s_\rho = [\![\texttt{If}\,(\texttt{Bin}\,(\texttt{Ne}\,, e_1, \texttt{Const}\ 0), s_t, s_f)]\!]^s_\rho$$

Noter les étiquettes sont « fraîches » (obtenues par `Gen.new_label`).

La boucle `while` est similaire.

$$[\![\mathtt{While}\,(\mathtt{Bin}\,(relop, e_1, e_2), s_l)]\!]^s_\rho =$$

$$\mathtt{Seq}\left[\begin{array}{l} \mathtt{Label}\ test; \\ \mathtt{Cjump}\,(relop, [\![e_1]\!]^e_\rho, [\![e_2]\!]^e_\rho, loop, fi); \\ \mathtt{Label}\ loop; [\![s_l]\!]^s_\rho; \mathtt{Jump}\ test; \\ \mathtt{Label}\ fi; \end{array}\right]$$

Les variables

Les environnements associent les noms des variables à, ...

- un temporaire t,

$$\begin{aligned}
[\![\texttt{Get}\ x]\!]^e_\rho &= \texttt{Temp}\ t\\
[\![\texttt{Set}\,(x, e)]\!]^s_\rho &= \texttt{Move_temp}\,(t, [\![e]\!]^e_\rho)
\end{aligned}$$

- une adresse mémoire a,

$$\begin{aligned}
[\![\texttt{Get}\ x]\!]^e_\rho &= \texttt{Mem}\ a\\
[\![\texttt{Set}\,(x, e)]\!]^s_\rho &= \texttt{Move_mem}\,(a, [\![e]\!]^e_\rho)
\end{aligned}$$

- un frame F (représentation ad-hoc des fonctions)

$$\begin{aligned}
[\![\texttt{Procedure_call}\,(f, [e_1; \ldots\ ; e_n])]\!]^s_\rho &= \texttt{Exp}\,(\texttt{Call}\,(F, [[\![e_1]\!]^e_\rho; \ldots\ ; [\![e_n]\!]^e_\rho]))\\
[\![\texttt{Function_call}\,(f, [e_1; \ldots\ ; e_n])]\!]^e_\rho &= \texttt{Call}\,(F, [[\![e_1]\!]^e_\rho; \ldots\ ; [\![e_n]\!]^e_\rho])
\end{aligned}$$

Fabriquer les environnents

Dans le cas simple de Pseudo-Pascal.

- La i-ième variable globale est rangée en mémoire, d'adresse :

$$\texttt{Plus}\,(\texttt{Const}\,(w * (i-1)), \texttt{Name}\ l_{\text{globals}})$$

  Autrement dit les variables globales sont en mémoire à la queue-leu-leu. L'adresse de cette zone est une étiquette définie dans le module `Frame`. On peut aussi décider qu'un registre Frame.global_register est réservé pour contenir cette adresse.

$$\texttt{Plus}\,(\texttt{Const}\,(w * (i-1)), \texttt{Temp}\ r_{\text{globals}})$$

- Les autres variables (variables locales, arguments) sont des temporaires.
- Les noms de fonction sont associés à une structure ad-hoc, nommée *frames*.

En pratique

Le type générique des environnements (module `Env`) est :

```
type ('a, 'b) environment
```

Les environnements utilisés pour compiler Pseudo-Pascal :

```
(case, Frame.frame) environment
```

Où case est du style :

```
type case =
| Tempo of Gen.temp       (* La variable est un temporaire *)
| Memoire of Code.expr    (* La variable est en mémoire *)
```

Introduction des temporaires

- Un temporaire « *frais* » pour chaque variable locale. (*i.e.* un temporaire obtenu par un appel à `Gen.new_temp`).
- Un temporaire par paramètre formel (et pour le résultat).
  - Pour ne pas dépendre des conventions d'appel à ce niveau.
  - Pour donner une chance d'aller en registre.

Les frames (représentation des fonctions) contiennent la liste des temporaires représentant leurs paramètres (et leur résultat éventuel).

Les frames (enfin)

C'est un point de rendez-vous dans tout le back-end.

```
type frame = {
    name : Gen.label;             (* Point d'entrée (prologue) *)
    return_label : Gen.label;     (* Adresse de l'épilogue *)
    args : Gen.temp list;         (* Temporaire des arguments *)
    result : Gen.temp option;     (* Temporaire du résultat (ou rien) *)
    mutable mysize : int;         (* Taille nécessaire sur la pile *)
  }
```

Mais le type des frames est abstrait, dans `frame.mli`, on aura la déclaration : **type** frame. En effet

- Le type frame peut changer (machine ciblée, langage compilé).
- Il est compliqué.

Interface (fichier `frame.mli`)

```
type frame
(* Création *)
val named_frame :
  string -> Pp.var_list -> Pp.type_expr option -> frame

(* Accéder aux informations *)
val frame_name : frame -> label
val frame_args : frame -> temp list
val frame_result : frame -> temp option
val frame_return : frame -> label
```

Compilation des programmes

Compiler une fonction

En deux étapes.

- Créer le frame.
- Compiler le corps :
  - Fabriquer le bon environnement.
  - Compiler le corps (une liste d'instructions) dans cet environnement.

```
val cfun :
  (access, Frame.frame) Env.environment ->
  string * Pp.definition -> Frame.frame * Code.stm
```

Compiler un programme

En quatre étapes :

- Fabriquer tous les frames (à cause des appels potentiellement récursifs).
- Fabriquer un environnement initial.
- Ajouter une fonction « principale ».
- Compiler toutes les fonctions.

```
type 'a procedure = Frame.frame * 'a
type 'a program =
    { number_of_globals : int;
      main : 'a procedure;
      procedures : 'a procedure list
    }

val program : Pp.program -> Code.stm program
```

Code intermédiaire, deuxième partie

$$\text{SA}\ \underbrace{\xrightarrow{\text{Générer du code}}\ \text{CI}\ \xrightarrow{\text{Canoniser Linéariser}}\ \text{CI}\ \xrightarrow{\text{Optimiser le contrôle}}}_{\text{Compilation}}\ \text{CI}$$

- Notre code n'est pas encore prêt pour la phase suivante (la *séléction*).
- On peut optimiser le contrôle (*e.g* éviter des sauts inutiles).

Notre code n'est pas prêt

En première approximation, notre code est *prêt pour la sélection* si sa traduction en code assembleur est simple et indépendante de l'ordre d'évaluation des expressions.

- On veut une liste d'instructions (supprimer les `Seq`).
- On veut des sauts conditionnels raisonables.

$$\texttt{Cjump}\,(relop, e_1, e_2, \ell_T, \ell_F); \texttt{Label}\ \ell_F$$

- Mais surtout, les expressions sont des arbres trop génériques.

Pouvoir de la sélection

Il est souhaitable que les expressions soient des arbres.

$$\texttt{Move}\,(t_0, \texttt{Bin}\,(\texttt{Plus}\,, e_1, e_2))$$

Est selectionnable de au moins deux façons :

$$\frac{\begin{array}{cc} \text{Code qui calcule } e_1 \text{ dans } r & \text{Code qui calcule } e_2 \text{ dans } r \\ \text{Code qui calcule } e_2 \text{ dans } r' & \text{Code qui calcule } e_1 \text{ dans } r' \end{array}}{\Downarrow}$$

$$t_0 \leftarrow r + r'$$

À quoi bon ? Par exemple, bonne utilisation des registres.

$$\mathcal{N}(e_1 + e_2) = \begin{cases} \max(\mathcal{N}(e_1), \mathcal{N}(e_2)) & \text{Si } \mathcal{N}(e_1) \neq \mathcal{N}(e_2) \\ \mathcal{N}(e_1) + 1 & \text{Sinon} \end{cases}$$

À condition de mettre en premier l'argument de $\mathcal{N}(e_i)$ maximal.

Sélection

`Bin (Plus , Bin (Plus , Temp` t_1`, Const 2), Bin (Plus , Temp` t_2`, Const 1))`

Peut donner lieu à deux selections

add t_3, t_1, 2	**add** t_3, t_2, 1
add t_4, t_2, 1	**add** t_4, t_1, 2
add t_5, t_3, t_4	**add** t_5, t_3, t_4

On peut laisser la sélection décider : il n'y aucune implication sémantique.

Sélection, autre exemple

$$\texttt{Bin}\,(\texttt{Plus}\,,\texttt{Call}\,(f,[e_1]),\texttt{Call}\,(g,[e_2]))$$

On ne peut pas laisser la sélection décider : les deux choix ont des sémantiques différentes (effets de bords dans les fonctions).

Comment s'en tirer :

- Ne pas (trop) spécifier l'ordre d'évaluation : une idée discutable.
  - Le résultat des programmes mal-écrits n'est pas intuitif.
  - Des programes optimisés/non optimisés peuvent donner des résultats différents.
  - C'est pas moderne.
- Tenir compte de l'ordre d'évaluation dans la sélection : une mauvaise idée.
  - C'est compliqué.
  - La sélection dépendrait alors du langage compilé !

Plus grave, la compilation des appels imbriqués

$$\texttt{Call}\,(f, \texttt{Call}\,(g, e_1), \texttt{Call}\,(h, e_2))$$

Ici c'est grave, la selection la plus naïve est incorrecte.

$$\begin{array}{ll}
\texttt{a0} \leftarrow e_1 & \#\ \text{argument de } g \\
\texttt{call } g & \#\ \text{le résultat de } g \ldots \\
\texttt{a0} \leftarrow \texttt{v0} & \#\ \text{est le premier argument de } f. \\
\texttt{a0} \leftarrow e_2 & \#\ \text{argument de } h \\
\ldots & \#
\end{array}$$

Notons qu'il est possible de s'en tirer dans la selection.

`a0` $\leftarrow e_1$	#	argument de g
`call` g	#	le résultat de $g\ldots$
`t0` $\leftarrow$ `v0`	#	« sauver » le premier argument de f.
`a0` $\leftarrow e_2$	#	argument de h
`call` h	#	
`a0` $\leftarrow$ `t0`	#	premier argument de f
`a1` $\leftarrow$ `v0`	#	second argument de f
`call` f	#	

Une fois de plus, c'est compliqué (et dupliqué dans chaque sélecteur).

Une solution définitive et modulaire

Avant la sélection, le code est « *canonisé* ».

Le code canonique est un code intermédiaire contraint :

- Il n'y a pas d'instructions Seq.
- Et surtout : les appels de fonctions sont contraints.

$$\texttt{Move}\,(t, \texttt{Call}\,(f, \ldots)) \qquad \texttt{Exp}\,(\texttt{Call}\,(f, \ldots))$$

- Idée : quelque soit la sélection des expressions, l'ordre d'évaluation est préservé.
- Il suffit de respecter l'ordre des instructions.

Un example de canonisation

$$\texttt{Seq}\,[\texttt{Move}\,(t_0, \texttt{Bin}\,(\texttt{Plus}\,, \texttt{Call}\,(f, e_1), \texttt{Call}\,(g, (\texttt{Call}\,(h, e_2)))));\ldots]$$

Se canonise en :

$$\texttt{Move}\,(t_1, e_1)$$

$$\texttt{Move}\,(t_2, \texttt{Call}\,(f, \texttt{Temp}\ t_1))$$

$$\texttt{Move}\,(t_3, e_2)$$

$$\texttt{Move}\,(t_4, \texttt{Call}\,(h, \texttt{Temp}\ t_3))$$

$$\texttt{Move}\,(t_5, \texttt{Call}\,(g, \texttt{Temp}\ t_4))$$

$$\texttt{Move}\,(t_0, \texttt{Bin}\,(\texttt{Plus}\,, \texttt{Temp}\ t_1, \texttt{Temp}\ t_5))$$

$$\ldots$$

Décrire la canonisation : l'arme absolue

Règle (de réécriture) $e \longrightarrow s \oplus c$. Une expression e se réduit un code s et une expression résiduelle c canoniques.

$\texttt{Const \_} \longrightarrow \oplus \texttt{ Const \_} \qquad \texttt{Temp \_} \longrightarrow \oplus \texttt{ Temp \_} \qquad \texttt{Name \_} \longrightarrow \oplus \texttt{ Name \_}$

$$\frac{e \longrightarrow s \oplus c}{\texttt{Mem } e \longrightarrow s \oplus \texttt{Mem } c}$$

$$\frac{e_1 \longrightarrow s_1 \oplus c_1 \qquad e_2 \longrightarrow s_2 \oplus c_2}{\texttt{Bin}\,(op, e_1, e_2) \longrightarrow s_1; \texttt{Move}\,(t, c_1); s_2 \oplus \texttt{Bin}\,(op, \texttt{Temp } t, c_2)}$$

$$\frac{e_1 \longrightarrow s_1 \oplus c_1 \qquad \cdots \qquad e_n \longrightarrow s_n \oplus c_n}{\begin{array}{c}\texttt{Call}\,(f, [e_1; \ldots\ ; e_n]) \longrightarrow \\ s_1; \texttt{Move}\,(t_1, c_1); \ldots\ ; s_n; \texttt{Move}\,(t_n, c_n); \\ \texttt{Move}\,(t_{n+1}, \texttt{Call}\,(f, [\texttt{Temp } t_1; \ldots\ ; \texttt{Temp } t_n])) \oplus \texttt{Temp } t_{n+1}\end{array}}$$

Arme absolue II

L'instruction s se canonise en la liste d'instructions s' $(s \longrightarrow s')$.

$$\frac{s_1 \longrightarrow s'_1 \qquad \cdots \qquad s_n \longrightarrow s'_n}{\texttt{Seq}\,[s_1; \ldots s_n] \longrightarrow s'_1; \cdots; s'_n} \qquad \frac{e \longrightarrow s \oplus c}{\texttt{Move}\,(t, e) \longrightarrow s; \texttt{Move}\,(t, c)}$$

$$\frac{e_1 \longrightarrow s_1 \oplus c_1 \qquad e_2 \longrightarrow s_2 \oplus c_2}{\texttt{Move\_mem}\,(e_1, e_2) \longrightarrow s_1; \texttt{Move}\,(t, c_1); s_2; \texttt{Move\_mem}\,(t, c_2)}$$

$$\frac{e_1 \longrightarrow s_1 \oplus c_1 \qquad e_2 \longrightarrow s_2 \oplus c_2}{\texttt{Cjump}\,(relop, e_1, e_2, \ell_T, \ell_F) \longrightarrow s_1; \texttt{Move}\,(t, c_1); s_2; \texttt{Cjump}\,(relop, t, c_2, \ell_T, \ell_F)}$$

On peut éviter quelques transferts :

$$\frac{e \longrightarrow s \oplus c}{\texttt{Move}\,(t, \texttt{Call}\,(f, e)) \longrightarrow s; \texttt{Move}\,(t, \texttt{Call}\,(f, c))}$$

Bref, on met à plat les Seq et on canonise les expressions.

Une arme trop puissante

$$\texttt{Move}\,(t_0, \texttt{Bin}\,(\texttt{Plus}\,, \texttt{Bin}\,(\texttt{Mult}\,, \texttt{Temp}\ t_1, \texttt{Const}\ 4), \texttt{Const}\ 7))$$

Se canonise en :

$$\texttt{Move}\,(t', \texttt{Bin}\,(\texttt{Mult}\,, \texttt{Temp}\ t_1, \texttt{Const}\ 4))$$

$$\texttt{Move}\,(t_0, \texttt{Bin}\,(\texttt{Plus}\,, \texttt{Temp}\ t', \texttt{Const}\ 7))$$

Et c'est bien domage car certains processeurs peuvent selectionner une seule instruction.

```
leal    7(,t1,4), t0
```

Donc, on cherche à garder des expressions en arbre le plus possible.

Commutation

Informellement, deux expressions commutent si on peut les évaluer n'importe comment sans mettre la sémantique en danger.

Alors la règle suivante semble correcte.

$$\frac{e_1 \longrightarrow s_1 \oplus c_1 \qquad e_2 \longrightarrow s_2 \oplus c_2 \qquad \boxed{e_1 \text{ et } e_2 \text{ commutent}}}{\mathtt{Bin}\,(op, e_1, e_2) \longrightarrow s_1; s_2 \oplus \mathtt{Bin}\,(op, c_1, c_2)}$$

Si on réfléchit à la correction :

- c_1 et c_2 commutent toujours (voir les règles).
- On veut seulement évaluer c_1 après s_2, la sémantique commandant le contraire.

$$\frac{e_1 \longrightarrow s_1 \oplus c_1 \qquad e_2 \longrightarrow s_2 \oplus c_2 \qquad \boxed{c_1 \text{ et } s_2 \text{ commutent}}}{\mathtt{Bin}\,(op, e_1, e_2) \longrightarrow s_1; s_2 \oplus \mathtt{Bin}\,(op, c_1, c_2)}$$

Exemples de commutation

On se pose donc la question de savoir si évaluer c_1 avant ou après s_2 est indifférent.

- Si c_1 est une constante ? s_2 quelconque.
- Si c_1 est un temporaire t ? s_2 n'écrit pas dans t.
- Si c_1 est `Bin` (op, e_1, e_2) ? s_2 commute avec e_1 et e_2.
- Si c_1 est un appel de fonction ? C'est un piège, c_1 est canonique.

Comment généraliser ?

> On commute si c_1 ne lit pas « ce qui » est écrit par s_2 :
>
> - temporaires,
> - et mémoire.

Autres occurences de la commutation

Par exemple, pour l'écriture en mémoire :

$$\frac{e_1 \longrightarrow s_1 \oplus c_1 \qquad e_2 \longrightarrow s_2 \oplus c_2 \qquad \boxed{c_1 \text{ et } s_2 \text{ commutent}}}{\texttt{Move\_mem}\,(e_1, e_2) \longrightarrow s_1; s_2; \texttt{Move\_mem}\,(c_1, c_2)}$$

Et surtout pour les arguments des fonctions :

$$[\,] \longrightarrow \oplus[\,]$$

$$\frac{e_1 \longrightarrow s_1 \oplus c_1 \qquad [e_2; \cdots; e_n] \longrightarrow s \oplus [c_2; \cdots; c_n]}{[e_1; e_2; \cdots; e_n] \longrightarrow s_1; \texttt{Move}\,(t_1, c_1); s \oplus [t_1; c_2; \cdots c_n]}$$

$$\frac{e_1 \longrightarrow s_1 \oplus c_1 \qquad [e_2; \cdots; e_n] \longrightarrow s \oplus [c_2; \cdots; c_n] \qquad \boxed{c_1 \text{ et } s \text{ commutent}}}{[e_1; e_2; \cdots; e_n] \longrightarrow s_1; s \oplus [c_1; c_2; \cdots c_n]}$$

Se débarasser du Cjump bi-étiquette

Facile, à l'aide d'une fenêtre se déplaçant sur le code. (Procédé « trou de serrure » ou *peephole*).

$$
\begin{array}{rcl}
\texttt{Jump}\ \ell; \texttt{Label}\ \ell & \longrightarrow & \texttt{Label}\ \ell \\
\texttt{Cjump}\,(relop, e_1, e_2, \ell_t, \ell_f); \texttt{Label}\ \ell_f & \longrightarrow & \texttt{Cjump}\,(relop, e_1, e_2, \ell_t, \ell_f); \texttt{Label}\ \ell_f \\
\texttt{Cjump}\,(relop, e_1, e_2, \ell_t, \ell_f); \texttt{Label}\ \ell_t & \longrightarrow & \texttt{Cjump}\,(\neg relop, e_1, e_2, \ell_f, \ell_t); \texttt{Label}\ \ell_t \\
\texttt{Cjump}\,(relop, e_1, e_2, \ell_t, \ell_f) & \longrightarrow & \texttt{Cjump}\,(relop, e_1, e_2, \ell_t, \ell); \texttt{Label}\ \ell; \texttt{Jump}\ \ell_f
\end{array}
$$

Bien, mais on voudrait aussi, quand c'est possible :

$$\texttt{Jump}\ \ell; \texttt{Label}\ \ell \quad \longrightarrow$$

C'est possible quand ℓ n'apparaît pas dans le code.

Optimisation du contrôle

Le code à plat s'y prête très mal. (Même si le Cjmp bi-étiquette explicite la totalité du contrôle).

On transforme le code en « *graphe de flot* ».

- Les sommets sont des blocs de bases (*basic blocks*).
  Des suites (maximales) d'instructions ne contenant ni étiquettes ni saut.
- Il y a un arc de b_1 à b_2 si b_1 se termine par un saut (possible) vers le début de b_1.
- Il y une entrée (un arc partant de rien) et une (des) sortie (un arc n'allant nulle part).

Structures de données pour le graphe flot

Les sommets du graphe : des blocs de base.

```
type basic_block =
  {enter:Gen.label ;   (* Étiquette d'entrée *)
   mutable succ:stm ; (* Dernière instruction (un saut) *)
   body:stm list ;}    (* Instructions du bloc *)
```

La liste des blocs suffit pour représenter le graphe de flot.

Par souci d'efficacité, on ajoute une association étiquette → bloc.

```
type graph = basic_block list * (Gen.label, basic_block) Hashtbl.t

let get_block (_,t) ℓ = Hashtbl.find t ℓ
```

Exemple de graphe de flot

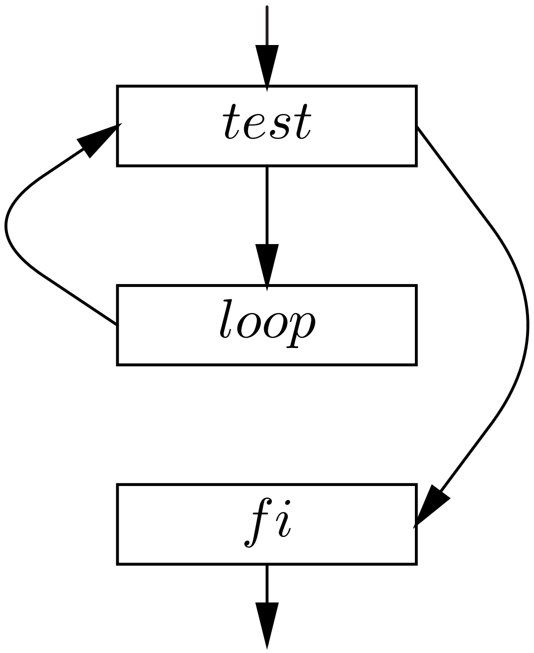
Le code :

$$\texttt{Label}\ test; \texttt{Cjump}\,(relop, [\![e_1]\!]^e_\rho, [\![e_2]\!]^e_\rho, loop, fi);$$

$$\texttt{Label}\ loop; \ldots\ ; \texttt{Jump}\ test;$$

$$\texttt{Label}\ fi;$$

Le graphe :



Application du graphe de flot

On compile l'instruction :

$$\texttt{If}\,(\texttt{Bin}\,(op_1, e_1, e_2), s_t, \texttt{If}\,(op_2, e_3, e_4, s_{ft}, s_{ff}))$$

Le code :

$\texttt{Label}\ start; \texttt{Cjump}\,(op_1, \_, \_, l_t, l_f);$

$\texttt{Label}\ l_t; \ldots\ ; \texttt{Jump}\ fi_1;$

$\texttt{Label}\ l_f;$

$\quad \texttt{Cjump}\,(op_2, \_, \_, l_{ft}, l_{ff});$

$\quad \texttt{Label}\ l_{ft}; \ldots\ ; \texttt{Jump}\ fi_2$

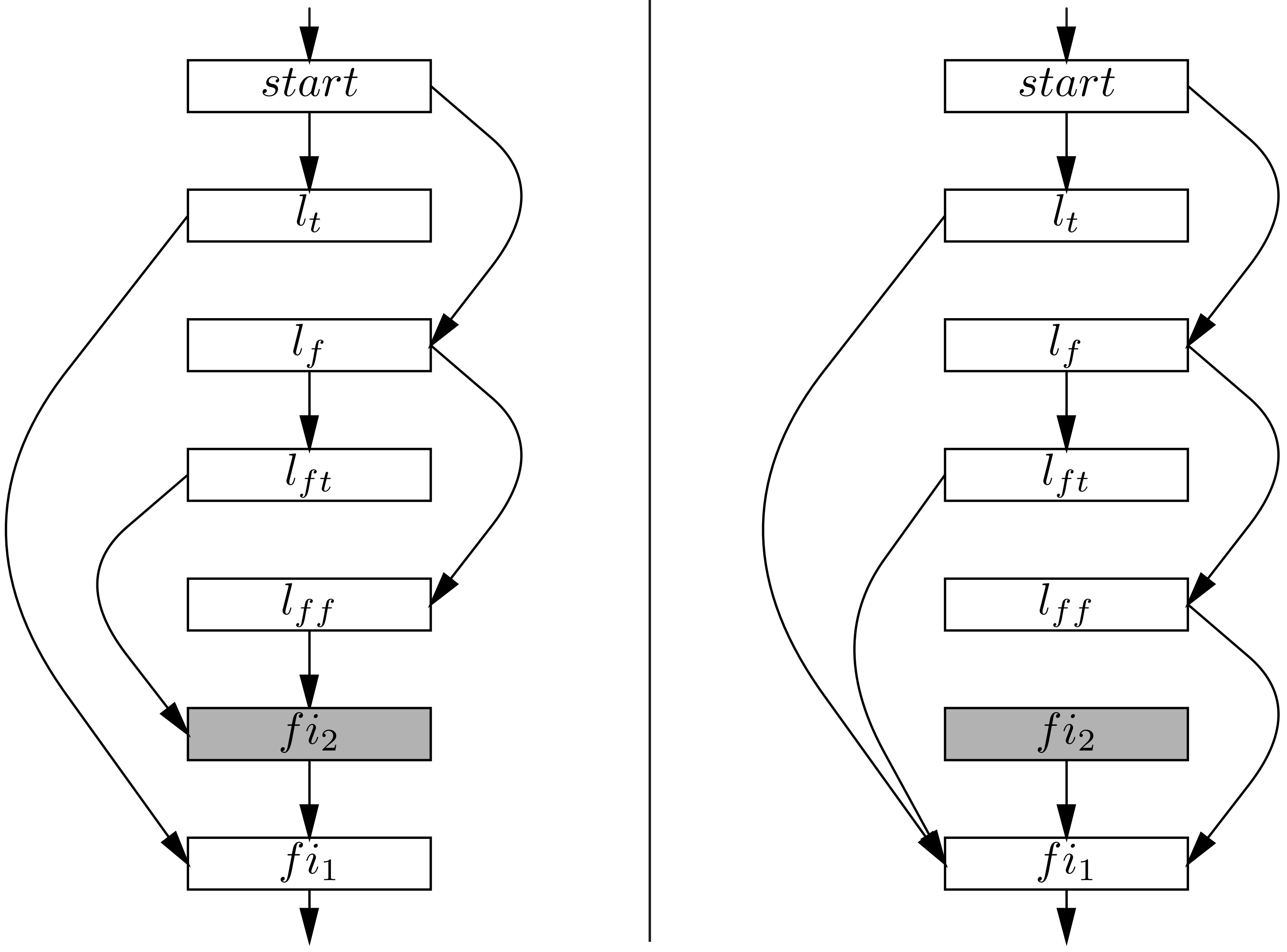
$\quad \texttt{Label}\ l_{ff}; \ldots\ ; \texttt{Jump}\ fi_2$

$\quad \texttt{Label}\ fi_2$

$\texttt{Jump}\ fi_1;$

$\texttt{Label}\ fi_1;$

Deux graphes équivalents



Optimisation du contrôle

Principe

1. Transformer le code (d'une fonction) en graphe de flot.
2. Transformer le graphe.
3. Transformer le graphe en code (et appliquer le trou de serrure).

Deux optimisation simples

- Court-circuiter les blocs vides.

  Éviter les sauts vers les sauts.

- Identifier le code mort.

  Supprimer du code inutile (cosmétique).

Court-circuit, en pratique

Court-circuiter une étiquette

```
let rec shorten_lab (_,t) lab =
  try
    let b = Hashtb.find t lab in
    match b with
    | {body=[]; succ=Jump olab} -> shorten_lab t olab
    | _ -> lab
  with
  | Not_found -> lab
```

Ne fonctionne pas toujours !

Boucle en cas de bloc vide bouclant sur lui-même.

Traiter un (les) bloc(s)

```
let shorten_block t b = match b.succ with
| Jump lab ->
    b.succ <- Jump (shorten_lab t lab)
| Cjump (op,e1,e2,lab1,lab2) ->
    b.succ <-
      Cjump (op, e1, e2, shorten_lab t lab1, shorten_lab t lab2)
| _ -> assert false

let shorten_blocks (blocks,t) =
  List.iter (shorten_block t) blocks,
  t
```

Pour identifier le code mort :

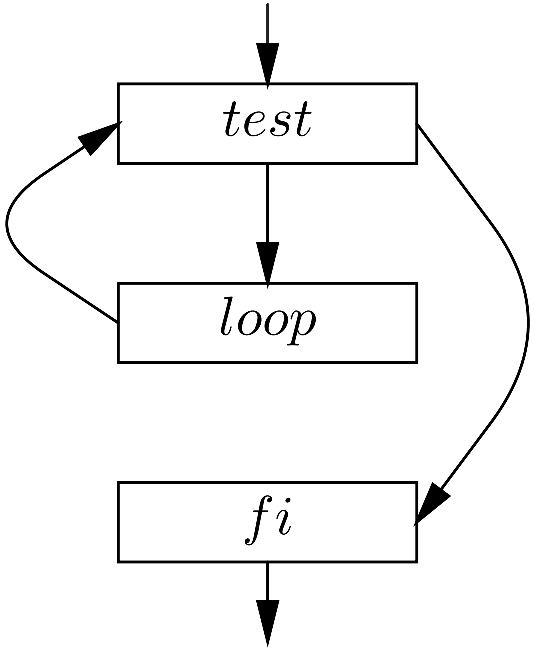
Un bête parcours de graphe (puis un filtrage de la liste de blocs).

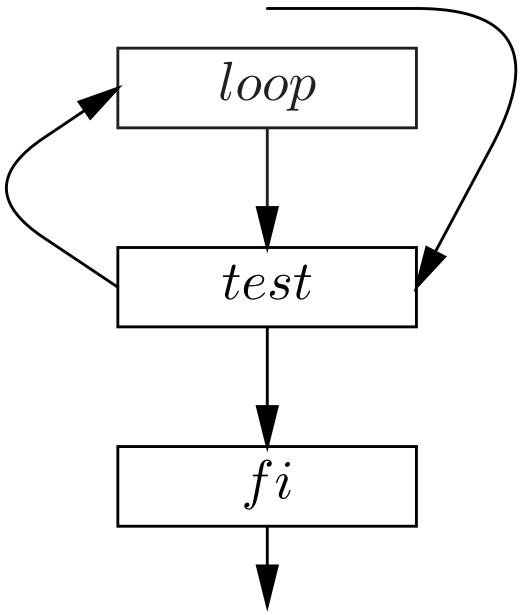
Encore plus fort

Modifier l'ordre de présentation des blocs, pour optimiser.

Sur le graphe de la boucle :

$\Rightarrow$





Code initial :

$$\texttt{Label}\ test; \texttt{Cjump}\,(relop, \_, \_, loop, fi);$$
$$\texttt{Label}\ loop; \ldots\ ; \texttt{Jump}\ test;$$
$$\texttt{Label}\ fi;$$

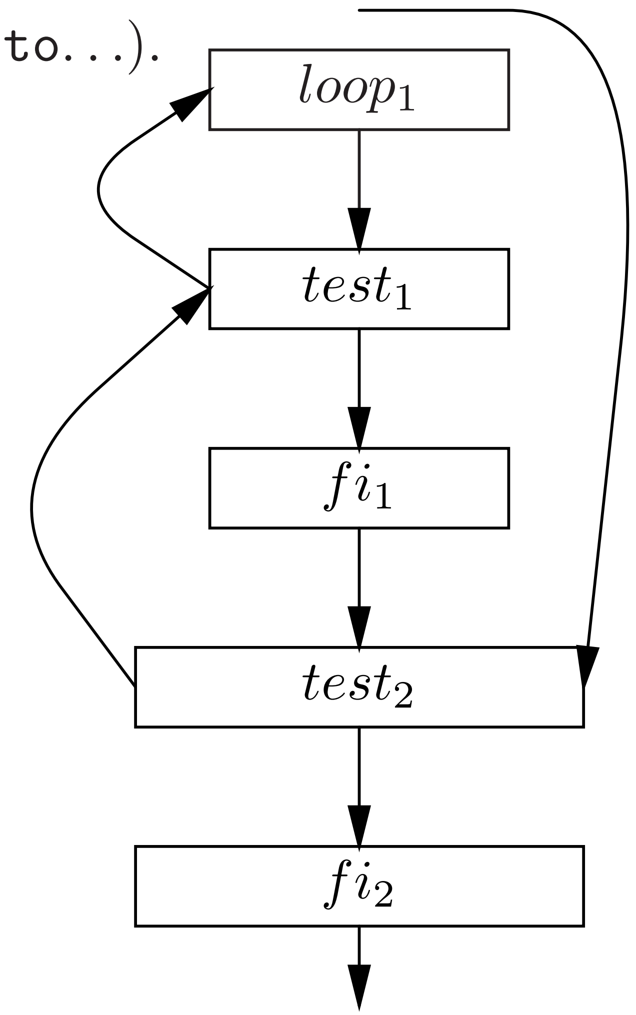
Code final :

$$\texttt{Jump}\ test; \texttt{Label}\ loop; \ldots\ ; \texttt{Jump}\ test;$$
$$\texttt{Label}\ test; \texttt{Cjump}\,(relop, \_, \_, loop, fi);$$
$$\texttt{Label}\ fi;$$

Est-ce bien utile ? Hum.

De plus en plus fort

Dans un graphe arbitraire, retrouver une structure de boucle imbriquée cachée (`goto`...).



Assez dur en pratique. Dommage.

Organisation de nos optimisations

On ne touchera pas à l'ordre des blocs (ne pas détruire la structure du code, déjà convenable, en voulant faire mieux).

1. Couper en blocs de base.
2. Produire le graphe.
3. Court-cicuiter les blocs vides.
4. Éliminer les blocs de *code mort*.
5. Fusionner le code des blocs mis bout-à bout.

Optimisations « trou de serrure » (*peephole*) et mise en place des test-and-branch style machine à la fin.

$$\begin{array}{rcl}
\texttt{Jump}\ l; \texttt{Label}\ l & \longrightarrow & \texttt{Label}\ l \\
\texttt{Cjump}\,(relop, e_1, e_2, l_t, l_f); \texttt{Label}\ l_f & \longrightarrow & \texttt{Cjump}\,(relop, e_1, e_2, l_t, l_f); \texttt{Label}\ l_f \\
\texttt{Cjump}\,(relop, e_1, e_2, l_t, l_f); \texttt{Label}\ l_t & \longrightarrow & \texttt{Cjump}\,(\neg relop, e_1, e_2, l_f, l_t); \texttt{Label}\ l_t \\
\texttt{Cjump}\,(relop, e_1, e_2, l_t, l_f) & \longrightarrow & \texttt{Cjump}\,(relop, e_1, e_2, l_t, l); \texttt{Label}\ l; \texttt{Jump}\ l_f
\end{array}$$

Les projets

Voir la page du cours. Modifications de notre compilateur `zyva`.

1. Recibler pour le Pentium.
2. Produire un système à bytecode.
3. Ajouter le passage par variable puis les fonctions locales.
4. votre projet