Introduction à la compilation

Christine Paulin-Mohring

Université Paris Sud

Master Informatique 2009-2010

Génération de code-I

1. Introduction
   - Rappel
   - Modèle d'exécution
2. Code intermédiaire
3. Compilation des appels de fonctions
4. Compilation d'un langage fonctionnel
5. Compilation d'un langage objet

Sommaire

1 Introduction
- Rappel
- Modèle d'exécution

2 Code intermédiaire

3 Compilation des appels de fonctions

4 Compilation d'un langage fonctionnel

5 Compilation d'un langage objet

Où en est-on ?

- L'analyse lexicale et syntaxique permet de construire un premier arbre de syntaxe abstraite et d'éliminer des programmes incorrects.
- L'analyse sémantique travaille par récurrence sur l'arbre de syntaxe abstraite pour collecter de nouvelles informations (portée, types, surcharge . . .) qui sont conservées dans l'arbre ou ailleurs.
  De nouvelles erreurs sont détectées.
- D'autres représentation comme le graphe de flots permettent de collecter des informations liées à l'exécution du programme.
- La génération de code va permettre d'engendrer un nouveau code plus proche du modèle d'exécution en machine.

Plan du cours

Génération de code pour une machine à pile

- *Introduction*: Les principes de base du modèle d'exécution
- *Code intermédiaire*
  - Instructions d'une machine à pile,
  - Cas de base,
  - Représentation des tableaux,
  - Compilation des expressions conditionnelles
- *Appel de procédures*: mode de passage de paramètres, tableaux d'activations (allocation des paramètres et des variables locales).
- *Allocation de registres*
- *Langages fonctionnels et objets*

Sommaire

1. Introduction
   - Rappel
   - Modèle d'exécution
2. Code intermédiaire
3. Compilation des appels de fonctions
4. Compilation d'un langage fonctionnel
5. Compilation d'un langage objet

Modèle d'exécution du code

- une partie de la mémoire est réservée aux instructions du programme
- un pointeur indique où on en est dans l'exécution
- la taille du code est connue à la compilation
- le code s'exécute de manière séquentielle :
  - sauf instruction explicite de saut, les instructions du programme sont exécutées l'une après l'autre.

Modèle d'exécution, représentation des données

- Les données du programme sont stockées soit dans la mémoire soit dans les registres de la machine.
- La mémoire est organisée en mots, on y accède par une adresse qui est représentée par un entier.
- Les valeurs simples sont stockées dans une unité de la mémoire.
- Les valeurs complexes sont stockées dans des cases consécutives de la mémoire (structures, tableaux), ou dans des structures chaînées (une liste est représentée par la valeur de son premier élément associée à l'adresse du reste de la liste).
- Les registres permettent d'accéder rapidement à des données simples.
- Le nombre de registres dépend de l'architecture de la machine et est limité.

Allocation

- Certaines données sont explicitement manipulées par le programme source par l'intermédiaire de variables
- D'autres données sont créées par le compilateur :
  - valeurs intermédiaires lors de l'exécution d'un calcul arithmétique
  - variables lors de l'appel de fonctions
  - valeurs objet, valeurs fonctionnelles ...
- Certaines données ont une durée de vie connue à la compilation: règles de portées du langage, variables vivantes.

Allocation-suite

- Les variables globales du programme peuvent être allouées à des adresses fixes, à condition de connaître leur taille.
- La gestion des variables locales des blocs et des procédures, ou bien le stockage de valeurs intermédiaires se prête bien à une gestion de l'allocation de la mémoire en pile (stack).
- D'autres données ont par contre une durée de vie qui n'est pas connue à la compilation.
  C'est le cas lorsque l'on manipule des pointeurs (variables dont la valeur est une adresse).
  Ces données vont être allouées en général dans une autre partie de la mémoire, organisée en tas (heap).

Mémoire

L'espace réservé dans le tas devra être libéré s'il n'est plus utilisé.

- commandes explicites pour libérer l'espace (C, Pascal)
- l'exécution du programme utilise une programme général de récupération de mémoire appelé communément gc pour garbage collector qui est traduit en glaneur de cellules ou ramasse-miettes (Caml, Java).

Sommaire

1. Introduction
2. Code intermédiaire
   - Introduction
   - Machine à pile
   - Expressions simples
   - Variables
   - Données complexes
   - Instructions conditionnelles
3. Compilation des appels de fonctions
4. Compilation d'un langage fonctionnel
5. Compilation d'un langage objet

Introduction

Code intermédiaire indépendant de la machine cible.

- factoriser une grande partie du travail de compilation;
- rendre le compilateur plus portable.

Le choix d'un langage intermédiaire est très important.

- assez riche pour permettre un codage aisé des opérations du langage sans créer de trop longues séquences de code
- assez limité pour que l'implantation finale ne soit pas trop coûteuse.

Nous expliquerons comment engendrer le code associé à des constructions de langage dans une machine à pile particulière.

Notations

Nous spécifierons une fonction *code* qui prend comme argument un arbre de syntaxe abstraite pour le langage et renvoie une suite d'instructions du code de la machine à pile.

Notations Notations concrètes pour représenter la syntaxe abstraite:

- Si $E ::= E_1 + E_2$ est une règle de grammaire nous écrirons
  $code(E_1 + E_2) = \ldots code(E_1) \ldots code(E_2)$
  spécifie la valeur de la fonction *code* sur un arbre de syntaxe abstraite correspondant à l'addition de deux expressions.
- Si C_1 et C_2 représentent des suites d'instructions alors $C_1 \mid C_2$ représente la suite d'instructions obtenue en concaténant C_2 à la suite de C_1.
- La liste vide d'instructions est représentée par $[]$.

Sommaire

1. Introduction
2. Code intermédiaire
   - Introduction
   - Machine à pile
   - Expressions simples
   - Variables
   - Données complexes
   - Instructions conditionnelles
3. Compilation des appels de fonctions
4. Compilation d'un langage fonctionnel
5. Compilation d'un langage objet

Principes de base de la machine à pile

Instructions

- On a une machine qui contient une zone de code, C,
- un registre *pc* (pointeur de code) contient l'adresse de l'instruction suivante à exécuter.
- Les instructions ont un nom suivi éventuellement de 1 ou 2 arguments.
- Ces arguments seront en général des entiers, des flottants ou bien des étiquettes symboliques indiquant des instructions du code qui seront traduites en des entiers lors d'une phase de préanalyse.

Machine à pile-suite

- La machine contient une pile P permettant de stocker des valeurs,
- un registre *sp* (stack pointer) pointe sur la première cellule libre de la pile.
- Un autre registre *gp* (global pointer) pointe sur la base de la pile, endroit où seront stockées les variables globales.
- Cette pile peut contenir des entiers, des flottants ou des adresses.
- Certaines données grosses en taille comme les chaînes de caractères sont allouées dans un espace supplémentaire, lorsqu'elle devra manipuler une chaîne, la pile se contentera de manipuler l'adresse de la chaîne.

Instructions

On présente pour chaque instruction de la machine, les modifications apportées à l'état de la pile et aux différents registres.

Exemple

Code	Pile	sp	pc	Condition
PUSHI n	$P[sp]:=n$	$sp+1$	$pc+1$	n est une valeur entière

- La commande **PUSHI** attend un argument n qui doit être un entier (sinon erreur d'exécution).
- Si cette exécution a lieu alors que la pile vaut P, que le registre de sommet de pile vaut sp et le compteur de programme vaut pc,
- alors après l'exécution du programme, la pile est modifiée (valeur n à l'adresse sp); les registres sp et pc sont incrémentés de 1.

Sommaire

1. Introduction
2. Code intermédiaire
   - Introduction
   - Machine à pile
   - Expressions simples
   - Variables
   - Données complexes
   - Instructions conditionnelles
3. Compilation des appels de fonctions
4. Compilation d'un langage fonctionnel
5. Compilation d'un langage objet

Expressions arithmétiques

La machine permet d'effectuer une opération arithmétique uniquement entre les deux valeurs au sommet de la pile, ces deux valeurs sont retirées de la pile et remplacées par une seule valeur représentant le résultat de l'opération.

Code	Pile	sp	pc	Condition
ADD	$P[sp-2] := P[sp-2] + P[sp-1]$	$sp-1$	$pc+1$	entier,entier
SUB	$P[sp-2] := P[sp-2] - P[sp-1]$	$sp-1$	$pc+1$	entier,entier
MUL	$P[sp-2] := P[sp-2] * P[sp-1]$	$sp-1$	$pc+1$	entier,entier
DIV	$P[sp-2] := P[sp-2] / P[sp-1]$	$sp-1$	$pc+1$	entier,entier erreur div 0

Comparaisons

Les valeurs booléennes peuvent être représentées par des entiers.

- 1 représente la valeur *vrai*
- 0 eprésente la valeur *faux*

Instructions

Code	Pile	sp	pc	Condition
INF	$\mathsf{P}[sp-2] \coloneqq$ si $\mathsf{P}[sp-2] < \mathsf{P}[sp-1]$ alors 1 sinon 0	$sp-1$	$pc+1$	entiers
INFEQ	$\mathsf{P}[sp-2] \coloneqq$ si $\mathsf{P}[sp-2] \leq \mathsf{P}[sp-1]$ alors 1 sinon 0	$sp-1$	$pc+1$	entiers
SUP	$\mathsf{P}[sp-2] \coloneqq$ si $\mathsf{P}[sp-2] > \mathsf{P}[sp-1]$ alors 1 sinon 0	$sp-1$	$pc+1$	entiers
SUPEQ	$\mathsf{P}[sp-2] \coloneqq$ si $\mathsf{P}[sp-2] \geq \mathsf{P}[sp-1]$ alors 1 sinon 0	$sp-1$	$pc+1$	entiers
EQUAL	$\mathsf{P}[sp-2] \coloneqq$ si $\mathsf{P}[sp-2] = \mathsf{P}[sp-1]$ alors 1 sinon 0	$sp-1$	$pc+1$	entiers, flottants, adresses

Flottants

On a un jeu d'instructions analogues mais qui agissent sur des flottants plutôt que sur des entiers.

Constantes

PUSHF

Opérations arithmétiques

FADD,FSUB,FMUL,FDIV

Comparaisons

FINF,FINFEQ,FSUP,FSUPEQ

Conversions

ITOF,FTOI

Génération de code

La représentation des expressions arithmétiques sous forme binaire permet d'engendrer simplement un code permettant le calcul des expressions arithmétiques.

Invariant

Exécution $code(E)$ à partir d'un état où $sp = n$:

- la valeur de l'expression E est calculée dans $\mathrm{P}[n]$
- les valeurs de $\mathrm{P}[m]$ pour $m < n$ n'ont pas été modifiées
- $sp = n + 1$

Code pour les expressions arithmétiques

Les expressions arithmétiques sont engendrées par la grammaire :

$$E ::= \texttt{entier} \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 * E_2 \mid E_1 / E_2$$

Le code correspondant est donné par :

$$
\begin{array}{ll}
\mathit{code}(\texttt{entier}) & = \textbf{PUSHI}\ \texttt{entier} \\
\mathit{code}(E_1 + E_2) & = \mathit{code}(E_1) \mid \mathit{code}(E_2) \mid \ \textbf{ADD} \\
\mathit{code}(E_1 - E_2) & = \mathit{code}(E_1) \mid \mathit{code}(E_2) \mid \ \textbf{SUB} \\
\mathit{code}(E_1 * E_2) & = \mathit{code}(E_1) \mid \mathit{code}(E_2) \mid \ \textbf{MUL} \\
\mathit{code}(E_1 / E_2) & = \mathit{code}(E_1) \mid \mathit{code}(E_2) \mid \ \textbf{DIV}
\end{array}
$$

Sommaire

1. Introduction
2. Code intermédiaire
   - Introduction
   - Machine à pile
   - Expressions simples
   - Variables
   - Données complexes
   - Instructions conditionnelles
3. Compilation des appels de fonctions
4. Compilation d'un langage fonctionnel
5. Compilation d'un langage objet

Variables globales

Deux instructions supplémentaires pour stocker des valeurs dans des variables :

- affecter une valeur calculée dans la pile à un espace réservé pour les valeurs globales;
- empiler au sommet de la pile une valeur contenue dans un variable globale.

Notation Si a est une adresse dans la pile et n un entier alors $a + n$ désigne l'adresse obtenue n emplacements au-dessus de a.

Code	Pile	sp	pc	Condition
PUSHG n	$P[sp] := P[gp+n]$	$sp+1$	$pc+1$	n entier $gp+n<sp$
STOREG n	$P[gp+n]:=P[sp-1]$	$sp-1$	$pc+1$	n entier $gp+n<sp$

Code pour les variables globales

$$E ::= \texttt{ident}$$

Un entier `adr(id)` est associé à chaque variable `id` au moment de l'analyse sémantique par exemple dans la table des symboles.

$$\mathit{code}(\text{id}) \quad = \textbf{PUSHG}\ \texttt{adr}(\text{id})$$

Réservation/Libération dans la pile

Commandes pour réserver ou libérer de l'espace sur la pile.

Code	Pile	sp	pc	Condition
PUSHN n	$P[sp+i] := 0 \quad \forall i. sp \leq i < sp+n$	$sp+n$	$pc+1$	n entier
POP n		$sp-n$	$pc+1$	n entier

À utiliser à l'initialisation du programme pour réserver la place nécessaire pour toutes les variables globales avant de démarrer les premiers calculs.
Le pointeur de sommet de pile initialement pointe sur la première case libre suivant les places réservées pour les globaux.

Affectations

Un langage dont les seules instructions (non-terminal I) sont des suites d'affectations.

Règles de grammaire

$$\begin{array}{lcl} I & ::= & \epsilon \\ I & ::= & I_1\ A; \\ A & ::= & \texttt{id}:=E \end{array}$$

Génération de code correspondante

$$\begin{array}{lcl} \mathit{code}(\epsilon) & = & [] \\ \mathit{code}(I_1\ A;) & = & \mathit{code}(I_1) \mid \mathit{code}(A) \\ \mathit{code}(\mathrm{id}:=E) & = & \mathit{code}(E) \mid \textbf{STOREG}\ \texttt{adr(id)} \end{array}$$

Sommaire

1. Introduction
2. Code intermédiaire
   - Introduction
   - Machine à pile
   - Expressions simples
   - Variables
   - Données complexes
   - Instructions conditionnelles
3. Compilation des appels de fonctions
4. Compilation d'un langage fonctionnel
5. Compilation d'un langage objet

Données composées

- On peut vouloir représenter des données structurées (produits, structures qui prendront plus d'un mot).
- On regarde ici le cas des produits :

$$E ::= (E_1, E_2)$$

- Suivant le langage de programmation, une valeur dans un type complexe peut être
  - les valeurs de l'objet sur la pile (structures en C)
  - une adresse d'un objet alloué dans le tas (ML, Java).
- Il faut préciser le mode de représentation des objets complexes

Paires représentées par valeur dans la pile

$$\begin{aligned}
\mathit{code}((E_1,E_2)) &= \mathit{code}(E_1) \mid \mathit{code}(E_2) \\
\mathit{code}(\mathrm{id}) &= \textbf{PUSHG}\ \texttt{adr(id)} \mid \textbf{PUSHG}\ \texttt{adr(id)} + 1 \\
&\quad \mid \ldots \mid \textbf{PUSHG}\ \texttt{adr(id)} + (k-1)
\end{aligned}$$

`adr`(id) représente l'adresse de base.
La taille de la variable dépend de son type, stockée dans la table des symboles.

$$\mathit{code}(\mathrm{id}{:=}E) = \mathit{code}(E) \mid \textbf{STOREG}\ \texttt{adr(id)}{+}(k{-}1) \mid \ldots \mid \textbf{STOREG}\ \texttt{adr(id)}$$

Tableaux

- Le nombre de cases de la pile dans lesquelles la donnée est représentée dépend du type de la donnée (entier, réel, tableau, record).
- On suppose la taille des données connue à la compilation et notée `taille(id)`.
- Les tableaux globaux peuvent être représentés par des suites de cases adjacentes dans la pile.

Tableaux-Calculs d'indice

- Si le tableau t a des indices compris entre m et M, démarre à l'adresse a et contient des éléments de taille k alors il occupe une place de $(M - m + 1) \times k$.
- Pour calculer l'adresse correspondante à une expression $t[E]$, il faut calculer la valeur n de E puis accéder à l'adresse $a + (n - m) \times k$.
- Si on connaît la valeur de m à la compilation alors on peut partiellement évaluer cette expression en précalculant $a - m \times k$ et gardant cette valeur dans la table des symboles `base(t)`.
- Il suffira ensuite d'effectuer l'opération $\texttt{base}(\texttt{t}) + n \times k$.

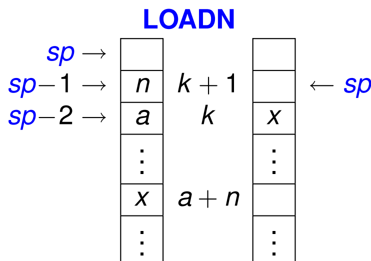
Tableaux-Fonctions d'accès

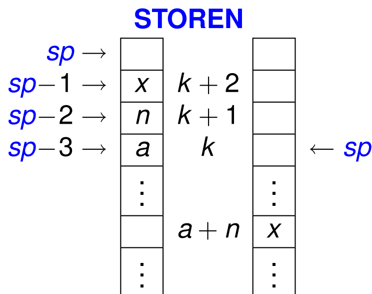
- Commandes **PUSHG,STOREG** permettent d'accéder à une adresse connue à la compilation.
- Accès à des tableaux : fonction de valeurs calculées à l'exécution.
- L'adresse à accéder dépend d'une adresse de base a (pas forcément connue statiquement -dans le cas de tableaux de taille variable) et d'un décalage n calculé à l'exécution.
- Commandes **LOADN** et **STOREN** analogues à **PUSHG,STOREG**, prennent les informations (adresses/décalage) dynamiquement sur la pile.

Code	Pile	sp	pc	Condition
LOADN	$P[sp-1]:=P[P[sp-2]+P[sp-1]]$	$sp-1$	$pc+1$	
STOREN	$P[P[sp-3]+P[sp-2]]:=P[sp-1]$	$sp-3$	$pc+1$	

Instructions LOADN/STOREN

Représentation graphique du comportement de ces deux instructions en matérialisant la pile dont on numérote les cases par les entiers $k, k+1 \ldots$:





Instructions alternatives

Une autre possibilité est de donner deux instructions

- **LOAD** accès à un élément de la pile par l'intermédiaire d'une adresse stockée dans la pile et un décalage constant
- **STORE** écriture d'un élément de la pile par l'intermédiaire d'une adresse stockée dans la pile et d'un décalage constant
- **PADD** calcul sur les adresses (ajout d'un entier à une adresse)

Code	Pile	sp	pc	Condition
LOAD n	$P[sp-1] := P[P[sp-1]+n]$	sp	$pc+1$	n entier
STORE n	$P[P[sp-2]+n] := P[sp-1]$	$sp-2$	$pc+1$	n entier
PADD	$P[sp-2] := P[sp-2]+P[sp-1]$	$sp-1$	$pc+1$	adresse+entier

Code - Expressions de tableau

On ajoute à notre langage des tableaux que l'on suppose unidimensionnels.

$$\begin{array}{lcl} E & ::= & \text{id}[E] \\ I & ::= & \text{id}[E_1] := E_2 \end{array}$$

Instruction **PUSHGP** qui permet de mettre sur la pile l'adresse du pointeur global dans la pile.

Code	Pile	*sp*	*pc*	Condition
PUSHGP	$P[sp]:=gp$	$sp+1$	$pc+1$	

Code - Expressions de tableau

Valeurs précalculées

- `adr`(id) adresse de base du tableau
- `taille`(id) taille d'un élément du tableau
- `base`(id)=`adr`(id)-m×`taille`(id)

Le code engendré est :

$$
\begin{array}{ll}
\textit{code}(\mathrm{id}[E]) & = \textbf{PUSHGP} \mid \textbf{PUSHI}\ \texttt{base}(\mathrm{id}) \\
 & \mid \textbf{PUSHI}\ \texttt{taille}(\mathrm{id}) \mid \textit{code}(E) \mid \textbf{MUL} \\
 & \mid \textbf{ADD} \mid \textbf{LOADN} \\
\textit{code}(\mathrm{id}[E_1]{:=}E_2) & = \textbf{PUSHGP} \mid \textbf{PUSHI}\ \texttt{base}(\mathrm{id}) \\
 & \mid \textbf{PUSHI}\ \texttt{taille}(\mathrm{id}) \mid \textit{code}(E_1) \mid \textbf{MUL} \\
 & \mid \textbf{ADD} \mid \textit{code}(E_2) \mid \textbf{STOREN}
\end{array}
$$

Tableaux bi-dimensionnels

- Tableaux bi-dimensionnels stockés en lignes ou en colonnes.
- Chaque dimension a pour indice minimal m_i et maximal M_i et pour taille $l_i = M_i - m_i + 1$.
- Stockage en ligne : on aura d'abord le tableau $t[m_1, *]$ (de taille l_2) puis $t[m_1 + 1, *]$ . . .
- L'élément $t[i_1, i_2]$ est stocké à l'adresse

$$a + ((i_1 - m_1) \times l_2 + (i_2 - m_2)) \times k$$

- Précalcul de l'expression : $(i_1 \times l_2 + i_2) \times k + a - (m_1 \times l_2 + m_2) \times k$
- Généralisation aux tableaux de dimension p.
- Dans le cas de tableau dont la taille n'est pas connue à la compilation, on ne pourra pas effectuer de précompilation.

Sommaire

1. Introduction
2. Code intermédiaire
   - Introduction
   - Machine à pile
   - Expressions simples
   - Variables
   - Données complexes
   - Instructions conditionnelles
3. Compilation des appels de fonctions
4. Compilation d'un langage fonctionnel
5. Compilation d'un langage objet

Instructions conditionnelles

- Les expressions conditionnelles et les boucles utilisent des commandes pour effectuer des sauts dans la zone d'instructions.
- Les instructions seront désignées par des adresses symboliques insérées dans le code:
  `label`: introduit un adresse symbolique `label` correspondant au numéro de l'instruction.

Code	Pile	sp	pc	Condition
JUMP `label`		sp	`label`	
JZ `label`		$sp-1$	si $P[sp-1]=0$ alors `label` sinon $pc+1$	

Convention pour la représentation des booléens : 0 peut représenter `faux` et pour `vrai`, on peut prendre les entiers positifs ou utiliser seulement 1.

Code conditionnel/boucles

$$I ::= \textbf{if } E \textbf{ then } I \textbf{ endif} \mid \textbf{if } E \textbf{ then } I_1 \textbf{ else } I_2 \textbf{ endif}$$
$$I ::= \textbf{while } E \textbf{ do } I \textbf{ done}$$

code(**if** E **then** I **endif**)
$= \mathit{code}(E) \mid$ **JZ** `new-suiv` $\mid \mathit{code}(I) \mid$ `new-suiv:`

code(**if** E **then** I_1 **else** I_2 **endif**)
$= \mathit{code}(E) \mid$ **JZ** `new-faux` $\mid \mathit{code}(I_1) \mid$ **JUMP** `new-suiv`
$\mid$ `new-faux:` $\mathit{code}(I_2) \mid$ `new-suiv:`

code(**while** E **do** I **done**)
$=$ `new-loop:` $\mathit{code}(E) \mid$ **JZ** `new-suiv` $\mid \mathit{code}(I)$
$\mid$ **JUMP** `new-loop` $\mid$ `new-suiv:`

Nouvelles étiquettes `new-faux`,`new-loop`, `new-suiv`.

Compilation d'expressions booléennes

$$
\begin{array}{lcl}
B & ::= & B_1 \textbf{ or } B_2 \mid B_1 \textbf{ and } B_2 \mid \textbf{not } B_1 \\
B & ::= & \textbf{vrai} \mid \textbf{faux} \\
B & ::= & E_1 \textbf{ relop } E_2
\end{array}
$$

Compiler les expressions booléennes comme des expressions arithmétiques. Empiler les valeurs 0 ou 1 en utilisant les opérations arithmétiques pour simuler les opérations booléennes.

$$
\begin{array}{lcl}
\mathit{code}(\textbf{vrai}) & = & \textbf{PUSHI } 1 \\
\mathit{code}(\textbf{faux}) & = & \textbf{PUSHI } 0 \\
\mathit{code}(B_1 \textbf{ and } B_2) & = & \mathit{code}(B_1) \mid \mathit{code}(B_2) \mid \textbf{MUL} \\
\mathit{code}(\textbf{not } B_1) & = & \textbf{PUSHI } 1 \mid \mathit{code}(B_1) \mid \textbf{SUB}
\end{array}
$$

Schéma de compilation conditionnel

Utilisation de conditions.

$$
\begin{aligned}
B_1 \textbf{ and } B_2 &= \textbf{if } B_1 \textbf{ then } B_2 \textbf{ else faux} \\
B_1 \textbf{ or } B_2 &= \textbf{if } B_1 \textbf{ then vrai else } B_2 \\
\textbf{not } B_1 &= \textbf{if } B_1 \textbf{ then faux else vrai}
\end{aligned}
$$

Dans un langage avec effets de bord, le fait de calculer ou non les sous-expressions d'une expression booléenne peut avoir des comportements très différents.

Schéma de compilation avec saut

- Les expressions booléennes servent souvent à des opérations de contrôle.
- On peut se donner a priori deux étiquettes `E-vrai` et `E-faux` et compiler l'expression booléenne sans calculer de valeur:
  - le résultat de l'exécution de E aboutit à $pc =$ `E-vrai` lorsque la valeur de E est **vrai** et à `E-faux` sinon.
- On définit donc une fonction *code-bool* qui prend comme argument l'expression booléenne et les deux étiquettes et qui renvoie le code de contrôle.

Code engendré

$$
\begin{array}{l}
\textit{code-bool}(\textbf{vrai}, e_v, e_f) = \textbf{JUMP}\ e_v \\
\textit{code-bool}(\textbf{faux}, e_v, e_f) = \textbf{JUMP}\ e_f \\
\textit{code-bool}(\textbf{not}\ B_1, e_v, e_f) = \textit{code-bool}(B_1, e_f, e_v) \\
\textit{code-bool}(B_1\ \textbf{or}\ B_2, e_v, e_f) = \\
\quad \textit{code-bool}(B_1, e_v, \texttt{new-e}) \mid \texttt{new-e}: \textit{code-bool}(B_2, e_v, e_f) \\
\textit{code-bool}(B_1\ \textbf{and}\ B_2, e_v, e_f) = \\
\quad \textit{code-bool}(B_1, \texttt{new-e}, e_f) \mid \texttt{new-e}: \textit{code-bool}(B_2, e_v, e_f) \\
\textit{code-bool}(E_1\ \textbf{relop}\ E_2, e_v, e_f) = \\
\quad \textit{code}(E_1) \mid \textit{code}(E_2) \mid \textit{code}(\textbf{relop}) \mid \textbf{JZ}\ e_f \mid \textbf{JUMP}\ e_v
\end{array}
$$

`new-e` : nouvelle étiquette
code(**relop**) l'instruction de comparaison pour l'opérateur **relop**.

Remarques

- On évite l'empilement de valeurs intermédiaires

$$
\begin{array}{l}
\textit{code}(\textbf{if } E \textbf{ then } I_1 \textbf{ else } I_2 \textbf{ endif}) = \\
\quad \textit{code-bool}(E, \texttt{new-vrai}, \texttt{new-faux}) \\
\quad \mid \texttt{new-vrai}: \textit{code}(I_1) \mid \textbf{JUMP}\ \texttt{new-suiv} \\
\quad \mid \texttt{new-faux}: \textit{code}(I_2) \mid \texttt{new-suiv}:
\end{array}
$$

Compilation branchements multiples

- Il s'agit d'expressions **switch** ou **case** permettant des branchements multiples suivant les différentes valeurs d'une expression
- ces valeurs, en général dans un type numérique, sont connues à la compilation:

$$\begin{array}{lll}\textbf{switch}\ E & \textbf{begin} & \textbf{case}\ V_1 : S_1 \\ & & \textbf{case}\ V_2 : S_2 \\ & & \ldots \\ & & \textbf{case}\ V_n : S_n \\ & \textbf{end} & \end{array}$$

Sémantique

Une sémantique naturelle est :

$$\begin{array}{lll}
\textbf{if } E = V_1 & \textbf{then} & S_1 \\
\textbf{else if } E = V_2 & \textbf{then} & S_2 \\
\ldots & & \\
\textbf{else if } E = V_n & \textbf{then} & S_n
\end{array}$$

Instruction **switch** dans C

- On exécute toutes les instructions $S_i; \ldots; S_n$ à partir du premier i tel que $E = V_i$
- La présence d'une instruction **break** à la fin de S_i permet de sortir du **switch**.

if $E = V_1$	**then**	**goto** l_1
else if $E = V_2$	**then**	**goto** l_2
...		
else if $E = V_n$	**then**	**goto** l_n
else	**goto** `fin`	
l_1 :		S_1
...		
l_n :		S_n
`fin:break` ;		

Génération de code

- On peut optimiser la compilation si les valeurs des V_i sont toutes distinctes dans un intervalle $[k+1, k+n]$.
- On crée dans le code une table de branchements qui débute par une étiquette `debut:` et qui comporte successivement les instructions : **JUMP** $l_1 \ldots$ **JUMP** l_n.
- Suivent ensuite les codes de chaque branche : l_i : $code(S_i)$ suivi éventuellement d'une instruction **JUMP** `fin`.

Saut indéxé

- Utilisation d’une instruction de saut indexé notée **JUMPI**.
- Cette instruction prend un label et renvoie le contrôle à l’instruction dont le numéro est la valeur numérique du label plus la valeur du sommet de la pile.

Code	Pile	*sp*	*pc*	Condition
JUMPI `label`		$sp-1$	$\texttt{label} + P[sp-1]$	

Génération de code-suite

- On calcule la valeur de E moins l'indice de base k du **switch** :

$$\mathit{code}(E) \mid \textbf{PUSHI}\ k \mid \textbf{SUB}$$

- On teste si la valeur obtenue est bien comprise entre 1 et n et on effectue le branchement indexé correspondant, sinon l'instruction est sautée.
- Comme la valeur en sommet de pile doit être utilisée pour deux comparaisons et le saut indéxé, il faut la dupliquer deux fois (instruction **DUP**).
- On obtient le code suivant :

$$\begin{array}{l} \textbf{DUP} \mid \textbf{DUP} \\ \mid \textbf{PUSHI}\ n \mid \textbf{INFEQ} \mid \textbf{JZ}\ \mathrm{fin} \\ \mid \textbf{PUSHI}\ 1 \mid \textbf{SUPEQ} \mid \textbf{JZ}\ \mathrm{fin} \\ \mid \textbf{JUMPI}\ \texttt{debut} \\ \mid \texttt{fin:} \end{array}$$

Génération de code II

1. Introduction
2. Code intermédiaire
3. Compilation des appels de fonctions
   - Exemple fonction récursive
   - Variables dans les procédures et fonctions
   - Tableau d'activation
   - Procédures emboitées
4. Compilation d'un langage fonctionnel
5. Compilation d'un langage objet

Introduction

- Compilation modulaire: code pour la fonction utilisant des paramètres formels, code de l'appel qui instancie ces paramètres.
- Différentes sémantiques pour le mode de liaison des paramètres
- Allocation dynamique de nouvelles variables (paramètres, variables locales)

Sommaire

1. Introduction
2. Code intermédiaire
3. Compilation des appels de fonctions
   - Exemple fonction récursive
   - Variables dans les procédures et fonctions
   - Tableau d'activation
   - Procédures emboitées
4. Compilation d'un langage fonctionnel
5. Compilation d'un langage objet

Exemple

Allocation de la mémoire pour les paramètres d'une fonction récursive avec appel par valeur

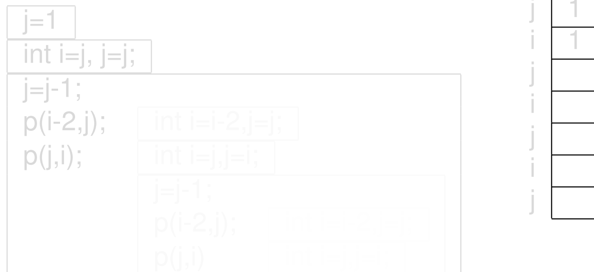
```
int j;
void p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
main () {read(j); p(j,j);}
```

- Le nombre d'exécutions de *p* dépend de la valeur lue en entrée.
- À chaque entrée dans la procédure *p* deux nouvelles variables *i* et *j* sont allouées.
- A la sortie de la procédure, les emplacement mémoires sont libérés.

Exemple-exécution

```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```

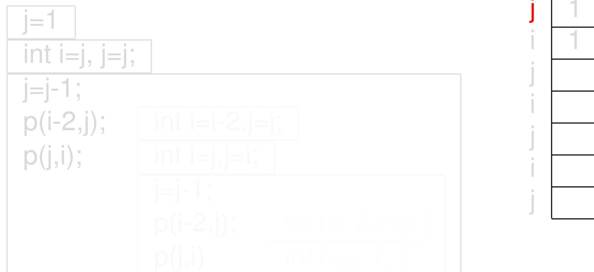
int j;
read(j);
p(j,j)



Exemple-exécution

```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```

int j;
read(j);
p(j,j)



j 1
i 1
j
i
j
i
j

Exemple-exécution

```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```

int j;
read(j);
p(j,j)

Exemple-exécution

```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```

```
int j;
read(j);    j=1
p(j,j)      int i=j, j=j;
            j=j-1;
            p(i-2,j);   int i=i-2,j=j;
            p(j,i);     int i=j,j=i;
                        j=j-1;
                        p(i-2,j);
                        p(j,i)
```

j	1
i	1
j	
i	
j	
i	
j	

Exemple-exécution

```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```

```
int j;
read(j);    j=1
p(j,j)      int i=j, j=j;
            j=j-1;
            p(i-2,j);   int i=i-2,j=j;
            p(j,i);     int i=j,j=i;
                        j=j-1;
                        p(i-2,j);   int i=i-2,j=j;
                        p(j,i)      int i=j,j=i;
```

j	1
i	1
j	
i	
j	
i	
j	

Exemple-exécution

```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```

```
int j;
read(j);      j=1
p(j,j)        int i=j, j=j;
              j=j-1;
              p(i-2,j);   int i=i-2,j=j;
              p(j,i);     int i=j,j=i;
                          j=j-1;
                          p(i-2,j);   int i=i-2,j=j;
                          p(j,i)      int i=j,j=i;
```

j	1
i	1
j	
i	
j	
i	
j	

Exemple-exécution

```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```

```
int j;
read(j);      j=1
p(j,j)        int i=j, j=j;
              j=j-1;
              p(i-2,j);   int i=i-2,j=j;
              p(j,i);     int i=j,j=i;
                          j=j-1;
                          p(i-2,j);   int i=i-2,j=j;
                          p(j,i)      int i=j,j=i;
```

j	1
i	1
j	
i	
j	
i	
j	

Exemple-exécution

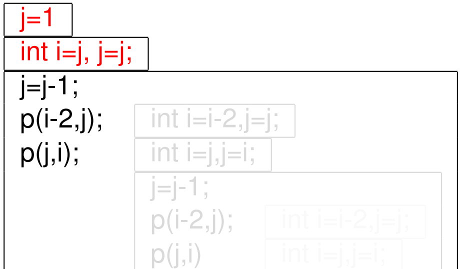
```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```

```
int j;
read(j);     j=1
p(j,j)       int i=j, j=j;
             j=j-1;
             p(i-2,j);   int i=i-2,j=j;
             p(j,i);     int i=j,j=i;
                         j=j-1;
                         p(i-2,j);   int i=i-2,j=j;
                         p(j,i)      int i=j,j=i;
```

j	1
i	1
j	1
i	
j	
i	
j	

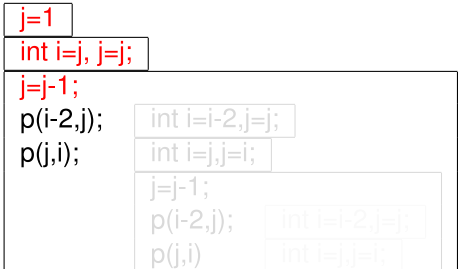
Exemple-exécution

```
p(i,j:int) {if (i+j>0) {j=j−1; p(i−2,j); p(j,i);}}
```



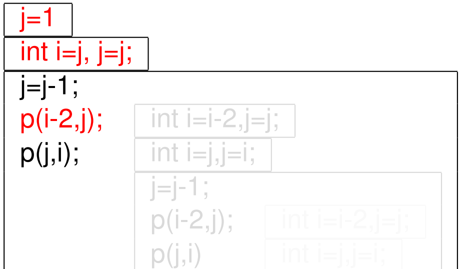
Exemple-exécution

```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```



Exemple-exécution

```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```

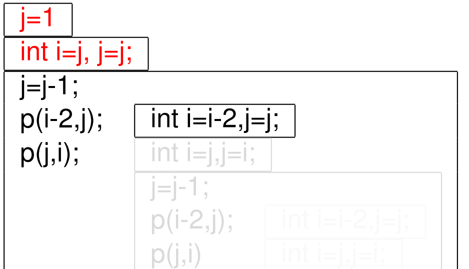


j	1
i	1
j	0
i	
j	
i	
j	

Exemple-exécution

```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```

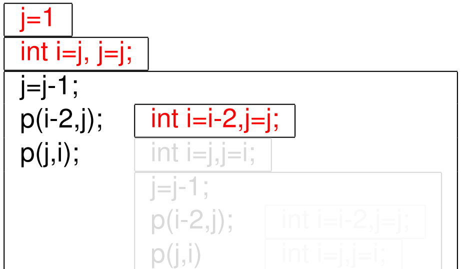
int j;
read(j);
p(j,j)



j	1
i	1
j	0
i	
j	
i	
j	

Exemple-exécution

```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```



j	1
i	1
j	0
i	
j	
i	
j	

Exemple-exécution

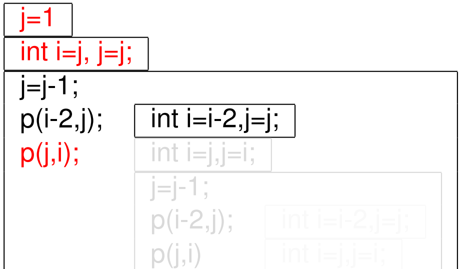
```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```

```
int j;
read(j);      j=1
p(j,j)        int i=j, j=j;
              j=j-1;
              p(i-2,j);   int i=i-2,j=j;
              p(j,i);     int i=j,j=i;
                          j=j-1;
                          p(i-2,j);   int i=i-2,j=j;
                          p(j,i)      int i=j,j=i;
```

j	1
i	1
j	0
i	-1
j	0
i	
j	

Exemple-exécution

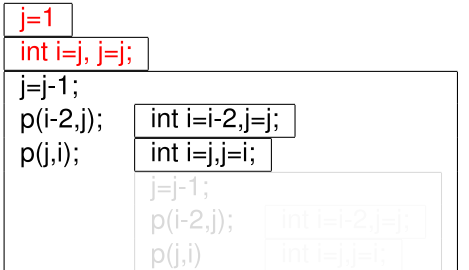
```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```



j	1
i	1
j	0
i	
j	
i	
j	

Exemple-exécution

```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```



j	1
i	1
j	0
i	
j	
i	
j	

Exemple-exécution

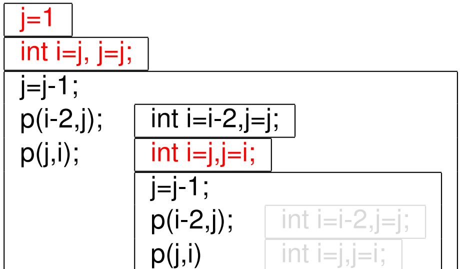
```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```

```
int j;
read(j);    j=1
p(j,j)      int i=j, j=j;
            j=j-1;
            p(i-2,j);   int i=i-2,j=j;
            p(j,i);     int i=j,j=i;
                        j=j-1;
                        p(i-2,j);   int i=i-2,j=j;
                        p(j,i)      int i=j,j=i;
```

j	1
i	1
j	0
i	0
j	1
i	
j	

Exemple-exécution

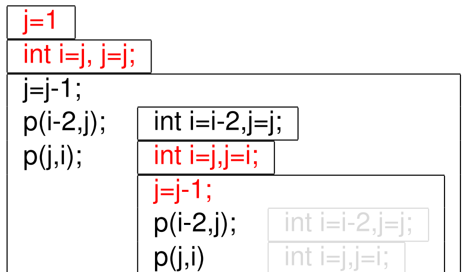
```
p(i,j:int) {if (i+j>0) {j=j−1; p(i−2,j); p(j,i);}}
```



j	1
i	1
j	0
i	0
j	1
i	
j	

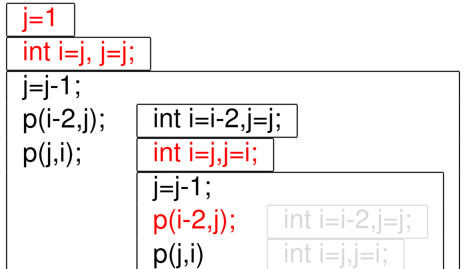
Exemple-exécution

```
p(i,j:int) {if (i+j>0) {j=j−1; p(i−2,j); p(j,i);}}
```



Exemple-exécution

```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```

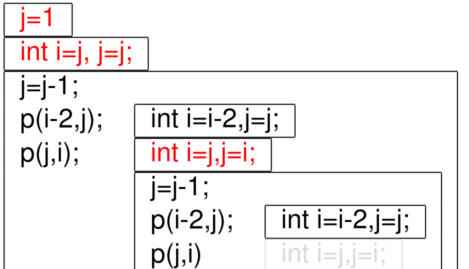


j	1
i	1
j	0
i	0
j	0
i	
j	

Exemple-exécution

```
p(i,j:int) {if (i+j>0) {j=j−1; p(i−2,j); p(j,i);}}
```

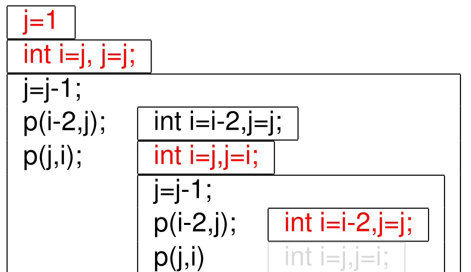
```
int j;
read(j);
p(j,j)
```



j	1
i	1
j	0
i	0
j	0
i	
j	

Exemple-exécution

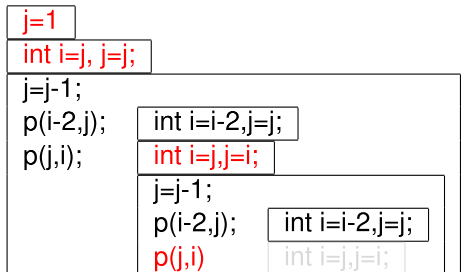
```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```



j	1
i	1
j	0
i	0
j	0
i	-2
j	0

Exemple-exécution

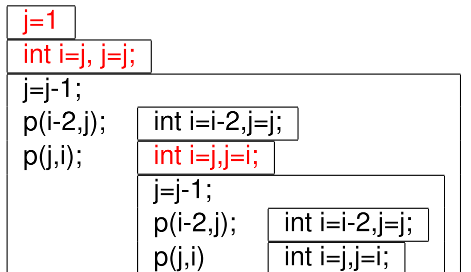
```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```



j	1
i	1
j	0
i	0
j	0
i	
j	

Exemple-exécution

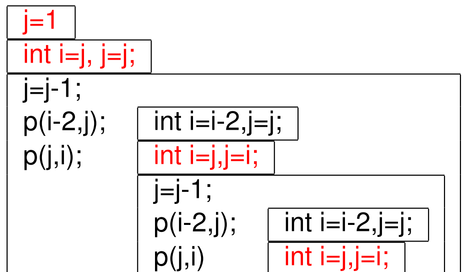
```
p(i,j:int) {if (i+j>0) {j=j−1; p(i−2,j); p(j,i);}}
```



j	1
i	1
j	0
i	0
j	0
i	
j	

Exemple-exécution

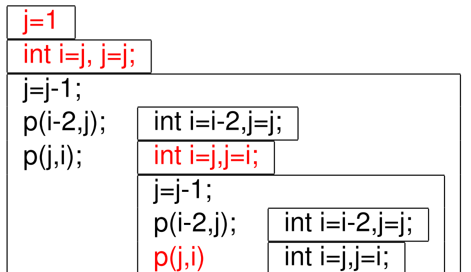
```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```



j	1
i	1
j	0
i	0
j	0
i	0
j	0

Exemple-exécution

```
p(i,j:int) {if (i+j>0) {j=j−1; p(i−2,j); p(j,i);}}
```

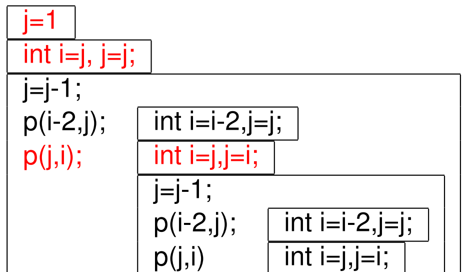


j	1
i	1
j	0
i	0
j	0
i	
j	

Exemple-exécution

```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```

int j;
read(j);
p(j,j)



j	1
i	1
j	0
i	
j	
i	
j	

Exemple-exécution

```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```

```
int j;
read(j);     j=1
p(j,j)       int i=j, j=j;
             j=j-1;
             p(i-2,j);   int i=i-2,j=j;
             p(j,i);     int i=j,j=i;
                         j=j-1;
                         p(i-2,j);   int i=i-2,j=j;
                         p(j,i)      int i=j,j=i;
```

j	1
i	1
j	
i	
j	
i	
j	

Sommaire

1. Introduction
2. Code intermédiaire
3. Compilation des appels de fonctions
   - Exemple fonction récursive
   - Variables dans les procédures et fonctions
   - Tableau d'activation
   - Procédures emboitées
4. Compilation d'un langage fonctionnel
5. Compilation d'un langage objet

Adresses mémoires et valeurs

- Certaines expressions de programme représentent des adresses mémoires
  - les variables introduites dans le programme;
  - les cases d'un tableau;
  - les composants d'une structure;
  - les pointeurs.
- Dans ces cases mémoires, sont stockées des valeurs.
- Certaines expressions de programme peuvent désigner des cases mémoires ou bien la valeur qui y est stockée.
  - La valeur gauche d'une expression représente la case mémoire à partir de laquelle est stockée l'objet (utilisée dans les affectations)
  - La valeur droite de l'expression représente la valeur de l'objet stockée dans la case mémoire.
- Le passage de paramètres aux procédures peut se faire en transmettant les valeurs gauches ou les valeurs droites des objets manipulés.

Variables-Environnement et état

- un environnement lie des noms à des adresses mémoires : fonction partielle qui associe une adresse à un identificateur.
- un état lie des adresses à des valeurs : une fonction partielle qui associe une valeur à une adresse.
- Lors d'une affectation d'une valeur à une variable, seul l'état change.
- Lors de l'appel d'une procédure comportant des paramètres ou des variables locales, l'environnement change.
  La variable introduite correspond à une nouvelle liaison entre un nom et une adresse.

Procédures et fonctions

- Une procédure a un nom, des paramètres, une partie de déclarations de variables ou de procédures locales et un corps.
- Une fonction est une procédure qui renvoie une valeur.

- Les paramètres formels sont des variables locales à la procédure qui seront instanciées lors de l'appel de la procédure par les paramètres effectifs.
- La procédure peut déclarer des variables locales qui seront initialisées dans le corps de la procédure.

Passage de paramètres

- Les paramètres formels de la procédure sont des variables qui sont initialisées lors de l'appel de la procédure.
- Il y a plusieurs manières d'effectuer cette initialisation.
- On suppose que l'on a une procédure p avec un paramètre formel x qui est appelée avec le paramètre effectif e.
- On examine différents mode de passage de ce paramètre.

Passage par valeur

Dans le passage de paramètre par valeur, x est une nouvelle variable allouée localement par la procédure dont la valeur est le résultat de l'évaluation de e.

- Après la fin de la procédure, la place mémoire allouée à la variable x est libérée.
- Les modifications apportées à x ne sont plus visibles.
- En l'absence de pointeurs, les seules variables modifiées sont les variables non locales à la procédure explicitement nommées dans les instructions du programme.
- Il est nécessaire de réserver une place proportionnelle à la taille du paramètre ce qui peut être coûteux dans le cas de tableaux.

Passage par référence ou par adresse

On calcule la valeur gauche de l'expression e
(si e n'a pas de valeur gauche on crée une variable que l'on initialise à la valeur droite de e et on utilise la valeur gauche de cette variable).

- La procédure alloue une variable x qui est initialisée par la valeur gauche de e.
- Toute référence à x dans le corps de la procédure est interprétée comme une opération sur l'objet situé à l'adresse stockée en x.
- Ce mode de passage occupe une place indépendante de la taille du paramètre (une adresse).

Remarques:

- En C, le passage par référence est explicitement programmé par le passage d'un pointeur (adresse mémoire) par valeur.
- En Java, le passage se fait par valeur mais les objets ont pour valeur une référence.

Passage par nom

- Remplacement textuel dans le corps de la procédure des paramètres formels par les paramètres effectifs.
- Mécanisme de macro qui peut provoquer des problèmes de capture.

Exemple

```
swap(int x; int y) {int z; z=x; x=y; y=z;}
```

Si *z* et *t* sont des variables globales du programme `swap`(z, t)

```
{int z; z=z; z=t; t=z;}
```

Renommage

- Renommer les variables locales.

```
swap(int x; int y) {int zz; zz=x; x=y; y=zz;}
```

- Cela ne suffit pas par exemple l'appel par nom de `swap`(i,a[i]) ne donne pas le résultat attendu.

```
{int z; z=i; i=a[i]; a[i]=z;}
```

- Méthode utile lors de la compilation de procédures de petite taille (coût de la gestion de l'appel est important).

Passage par copy-restore

- Lors de l'appel de la procédure la valeur droite de *e* sert à initialiser la variable *x*.
- À la sortie de la procédure la valeur droite de *x* sert à mettre à jour la valeur gauche de *e*.
- Des comportements parfois différents du passage par référence.

```
int a;
p (int x) {x=2; a=0;}
main () {a=1;p(a);write(a);}
```

Equivalent en appel par référence à

```
{a=1; a=2; a=0; write(a);}
```

Equivalent en copy-restore à

```
{a=1; {int x=a; x=2; a=0; a=x}; write(a);}
```

Évaluation stricte/paresseuse

Dans les langages fonctionnels, on distingue les langages stricts des langages dits paresseux.

- Langage strict : les valeurs des arguments sont calculées avant d'être passées en paramètre à une fonction (CAML ou SML).
- Langage paresseux : l'expression passée en argument à une fonction f sera évaluée seulement si f a besoin de cette valeur (on parle d'appel par nécessité, Haskell).
- L'évaluation paresseuse se combine mal avec les effets de bord (difficulté de controler à quel moment l'expression sera évaluée).

Évaluation paresseuse

- Définition $f(x) = t$, on veut calculer $f(e)$
- Lorsque l'évaluation de t nécessitera la valeur de x, le calcul de e sera effectué.
- Si t utilise plusieurs fois x, le calcul sera effectué une seule fois.
- L'expression e vient avec son environnement (les valeurs des variables utilisées par e) ce qui évite les problèmes de capture.
- Mécanisme différent de celui de remplacement textuel (macros).

Sommaire

1. Introduction
2. Code intermédiaire
3. Compilation des appels de fonctions
   - Exemple fonction récursive
   - Variables dans les procédures et fonctions
   - Tableau d'activation
   - Procédures emboitées
4. Compilation d'un langage fonctionnel
5. Compilation d'un langage objet

Compilation des fonctions et procédures

- Déclaration $f(x) = e$
- Appel $f(a)$ correspond à **let** $x = a$ **in** e
- Compiler le code de e en faisant une hypothèse sur l'emplacement de x
- Placer a à l'emplacement souhaité
- Sémantique de $x = a$
  - valeur de a
  - code de a
  - emplacement mémoire de a
  - ...

Procédures et fonctions-suite

- On ne contrôle pas le nombre d'appels d'une procédure ni le moment où celle-ci sera appelée.
- Il n'est pas possible de donner statiquement les adresses des variables apparaissant dans la procédure.
- Ces adresses pourront être calculées relativement à l'état de la pile au moment de l'appel de la procédure.
- L'espace mémoire alloué pour les paramètres et les variables locales est libéré lorsque l'on sort de la procédure.

Organisation de la mémoire

Dans un appel de procédure ou de fonction, la mémoire est organisée en tableaux d'activation.

- Le tableau d'activation est une portion de la mémoire qui est allouée à l'appel de la procédure et libérée au retour.
- Il contient les informations nécessaires à l'exécution du programme (paramètres, variables locales).
- Il sauvegarde les données qui devront être restaurées à la sortie de la procédure.

Afin de faciliter la communication entre des codes compilés à partir de langages distincts (par exemple pour appeler des procédures de bas niveau à partir d'un langage de haut niveau), il n'est pas rare qu'un format de tableau d'activation pour une architecture donnée soit particulièrement recommandé.

Données à conserver

Lors d'un appel de procédure:

- le contrôle du code est modifié:
  Retour normal de la procédure (pas d'échappement en erreur ou d'instruction de type `goto`) : la suite de l'exécution doit se poursuivre à l'instruction suivant l'instruction de l'appel.
- Le compteur d'instruction doit donc être sauvegardé à chaque appel de procédure.
- Les données locales à la procédure s'organisent dans la pile à partir d'une adresse sur le bloc d'activation (appelée framepointer) qui est déterminée à l'exécution et sauvée dans un registre (noté *fp*).
- Lorsqu'une nouvelle procédure est appelée, cette valeur change, il est donc nécessaire de sauvegarder la valeur courante qui pourra être restaurée à la fin de la procédure.

Appelant-Appelé

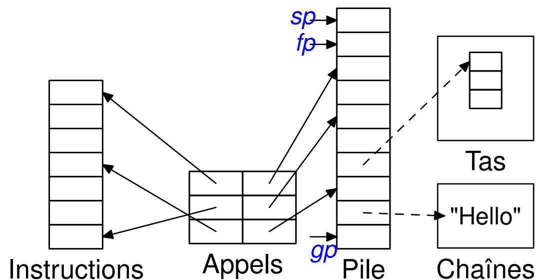
- Les opérations à effectuer lors d'un appel de procédure se partagent entre l'appelant et l'appelé.
- Le code géré par l'appelant doit être écrit pour chaque appel tandis que celui écrit dans l'appelé n'apparait qu'une seule fois.
- L'appelant effectue la réservation pour la valeur de retour dans le cas de fonctions et évalue les paramètres effectifs de la procédure.
- L'appelé initialise ses données locales et commence l'exécution.
- Au moment du retour, l'appelé place éventuellement le résultat de l'évaluation à l'endroit réservé par l'appelant et restaure les registres.

L'appelant et l'appelé doivent avoir une vision cohérente de l'organisation de la mémoire

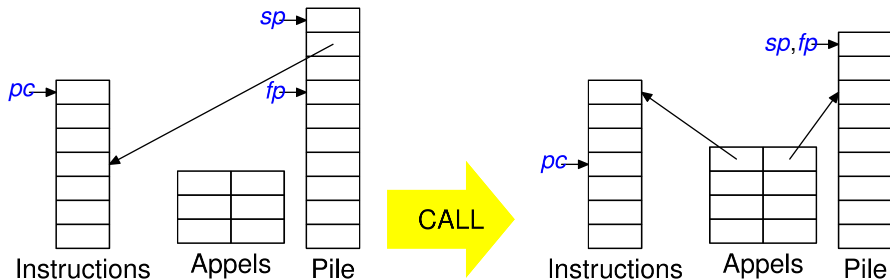
Appels de procédure : **CALL,RETURN**

- L'appel de procédure a deux effets:
  le premier est de modifier le cours d'exécution des instructions
  le second est de créer un espace pour des données locales en affectant le pointeur *fp*.
- L'instruction **CALL** prend comme argument une adresse dans le code d'instructions (en argument ou sur la pile).
- Le compteur d'instructions *pc* se place alors à cette adresse, son ancienne valeur est sauvegardée.
- **CALL** positionne le registre *fp* (adresse du bloc d'activation) à la valeur courante de *sp*.
- L'instruction **RETURN** retrouve l'ancienne valeur du compteur d'instructions et se place à l'instruction suivante. Elle repositionne *sp* à la valeur courante de *fp* et restaure l'ancien *fp*.

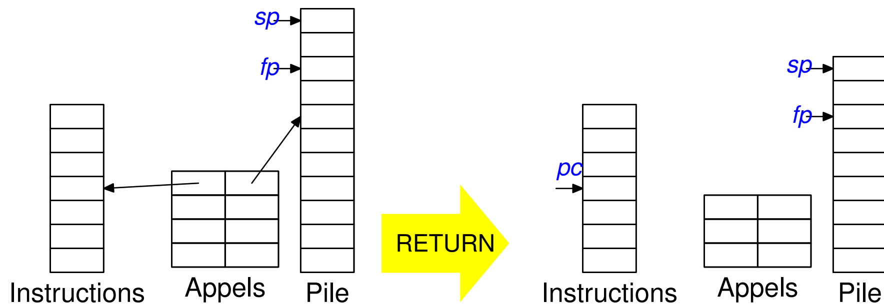
Organisation de la machine



Fonctionnement de **CALL**



Fonctionnement de **RETURN**



Sous-routines

- Réutiliser à plusieurs endroits la même séquence de code I.
- On isole cette partie du code et on lui donne une étiquette.
- Nous ajoutons des définitions et appels de routines dans notre langage :

$$\begin{array}{lcl} D & ::= & \textbf{proc}\ p;\ \textbf{begin}\ I\ \textbf{end} \\ I & ::= & \textbf{call}\ p \end{array}$$

- *label-p* est une étiquette unique associée à la procédure p.

$$\begin{array}{ll} code(\textbf{proc}\ p;\ \textbf{begin}\ I\ \textbf{end}) & = \texttt{label-p:}code(I) \mid \textbf{RETURN} \\ code(\textbf{call}\ p) & = \textbf{CALL}\ \texttt{label-p} \end{array}$$

Procédures avec paramètres

- Si la procédure a des paramètres alors elle alloue dans la pile la place pour ces variables.
- Le registre *fp* est à jour au début de l'appel de procédure et permet de référencer les valeurs locales.
- A la sortie de la procédure l'espace est libéré.
- Notre langage contient des instructions pour manipuler des données référencées à partir du pointeur *fp*.
- Aux instructions **PUSHGP, STOREG, PUSHG** correspondent les instructions **PUSHFP, STOREL, PUSHL** qui ont le même comportement en remplaçant *gp* par *fp*.

Tableau d'activation

Le tableau d'activation va être constitué :

- des paramètres de la procédure qui sont instanciés à l'appel
- des emplacements pour les variables locales.

$$
\begin{array}{lcl}
\mathit{Vs} & ::= & \mathit{Vs}\ V; \mid \epsilon \\
V & ::= & T\ \mathrm{id} \\
D & ::= & \mathrm{id}(\mathit{Vs})\ \{\mathit{Vs}\ I\ \} \\
 & \mid & T\ \mathrm{id}(\mathit{Vs})\ \{\mathit{Vs}\ I\ \textbf{return}\ E;\}
\end{array}
$$

Exemple

```
int f (int x; int y) { int z; z=x*x+y*y; return 2*z;}
```

Lors de l'appel $f(e_1, e_2)$,

- l'appelant réserve la place pour la valeur de retour et calcule les valeurs de e_1 et de e_2 sur la pile,
- puis il appelle **CALL** *f* ce qui positionne *fp* et donne la main à l'appelé.
- L'appelant alloue la variable *z* et effectue les calculs,
- il place la valeur de retour à l'endroit réservé avant d'appeler l'instruction **RETURN**.
- L'appelant doit libérer la place occupée par les paramètres.

Code

Tableau d'activation de f:

z		$\leftarrow fp$
y	e_2	$fp-1$
x	e_1	$fp-2$
`retour`		$fp-3$

Code de l'appelant:

PUSHN 1 | $code(e_1)$ | $code(e_2)$ | **CALL** `label-f` | **POP** 2

Code de l'appelé:

```
label-f: PUSHN 1
         PUSHL -2 | DUP | MUL |        //x * x
         PUSHL -1 | DUP | MUL |        //y * y
         ADD | STOREL 0 |              //z = x * x + y * y
         PUSHI 2 | PUSHL 0 | MUL |     //2 * z
         STOREL -3 | RETURN
```

Décalage

On décide d'une stratégie pour le placement des paramètres et variables locales dans le tableau d'activation des procédures.

- Les paramètres formels ont des décalages négatifs
- Les variables locales ont des décalages positifs.

Organisation du tableau d'activation

Pour chaque fonction ou procédure *f*, on peut stocker :

- `nretour`(f) la taille de la valeur de retour
- `nparams`(f) la taille des paramètres
- `nvars`(f) la taille des variables locales

Pour chaque variable *x*, on doit stocker :

- `islocal`(x) booléen vrai si la variable est locale.
- `decal`(x) entier indiquant la position relative par rapport à *gp* ou *fp* où est stockée la variable.
- `taille`(x) si les variables peuvent avoir une taille plus grande que 1

Code : cas d'une fonction avec passage par valeur

Appelant

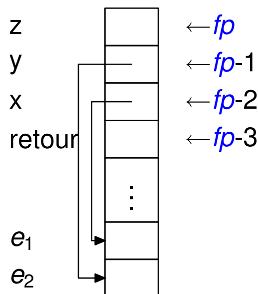
$$\mathit{code}(f(e_1,\ldots,e_n)) = \textbf{PUSHN}\ \texttt{nretour}(f) \mid \quad \text{// valeur retour}$$
$$\mathit{code}(e_1)\ldots\mathit{code}(e_n) \mid \quad \text{// empile les argts}$$
$$\textbf{CALL}\ f \mid \quad \text{// appelle } f$$
$$\textbf{POP}\ \texttt{nparams}(f) \quad \text{// libère la mémoire}$$

Appelé

$$\mathit{code}(T\ f(T_1\ x_1;\ldots T_n\ x_n)\{U_1\ z_1;\ldots U_p\ z_p\ I\ \textbf{return}\ E\}) =$$
$$\textbf{PUSHN}\ \texttt{nvars}(f) \mid \quad \text{// réserve place variables}$$
$$\mathit{code}(I) \mid \quad \text{// code instructions}$$
$$\mathit{code}(E) \mid \quad \text{// code valeur retour}$$
$$\textbf{STOREL}\ -\texttt{nparams}(f)-1 \mid \quad \text{stocke valeur retour}$$
$$\ldots \mid \textbf{STOREL}\ -\texttt{nparams}(f)-\texttt{nretour}(f) \mid$$
$$\textbf{RETURN} \quad \text{// Retour}$$

Passage par référence-exemple

Dans le cas où une variable est passée par référence, c'est son adresse qui est stockée dans le tableau d'activation.



Fonction qui calcule la valeur gauche d'une expression

Code pour les variables

Code appelant

PUSHN 1 | *code-gauche*(e_1) | *code-gauche*(e_2) | **CALL** `label-f` | **POP** 2

Code de l'appelé:

```
label-f: PUSHN1
         PUSHL −2 | LOAD 0 |           // x
         DUP | MUL |                   // x*x
         PUSHL −1 | LOAD 0 |           // y
         DUP | MUL |                   // y * y
         ADD | STOREL 0 |              // z=x*x+y * y
         PUSHI 2 | PUSHL 0 | MUL |     // 2*z
         STOREL −3 | RETURN
```

Code pour les variables

$code(x) = \textbf{PUSHG}\ \texttt{decal}(x)$ si $\neg$ `islocal(x)`
$code(x) = \textbf{PUSHL}\ \texttt{decal}(x)$ si `islocal(x)`, par valeur
$code(x) = \textbf{PUSHL}\ \texttt{decal}(x) \mid \textbf{LOAD}\ 0$ si `islocal(x)`, par référence

$code\text{-}gauche(x) = \textbf{PUSHFP} \mid \textbf{PUSHI}\ \texttt{decal}(x) \mid \textbf{PADD}$ si `islocal(x)`
$code\text{-}gauche(x) = \textbf{PUSHGP} \mid \textbf{PUSHI}\ \texttt{decal}(x) \mid \textbf{PADD}$ si $\neg$ `islocal(x)`
$code\text{-}gauche(x[E]) = code\text{-}gauche(x) \mid code(E) \mid \textbf{PADD}$

Sommaire

1. Introduction
2. Code intermédiaire
3. Compilation des appels de fonctions
   - Exemple fonction récursive
   - Variables dans les procédures et fonctions
   - Tableau d'activation
   - Procédures emboitées
4. Compilation d'un langage fonctionnel
5. Compilation d'un langage objet

Introduction

- Dans les langages comme Pascal ou CAML (contrairement à C ou Java), il est possible de définir des fonctions dans d'autres fonctions.
- Une fonction peut alors accéder à:
  - des variables globales : allouées à un emplacement fixe;
  - ses paramètres et variables locales : allouées dans son propre tableau d'activation;
  - les paramètres ou variables locales d'une des procédures dans laquelle elle est définie et qui sont allouées dans le tableau d'activation de cette procédure .

Arbre de niveau des déclarations

- On suppose que l'on a un langage fonctionnel ou impératif dans lequel procédures, fonctions et variables peuvent être arbitrairement imbriquées.
- La portée des variables se fait de manière statique suivant les règles habituelles de visibilité.

Exemple

```
let main a =
  let e = read_float () in
  let iter z =
    let stop x = abs (a − x ∗ x) < e in
    let next () = (z + (a / z)) / 2 in
    if stop z then z else
    let z' = next () in iter z'
  in iter 1
```

Exemple d'arbre de niveau de déclarations

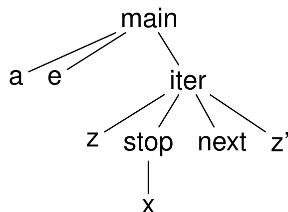
Niveaux

0 : main

1 : a, e, `iter`

2 : z, `stop`, `next`, z'

3 : x



Visibilité

main : `main`, a, e, `iter`

iter : `main`, a, e, `iter`, z, `stop`, `next`, z'

stop : `main`, a, e, `iter`, z, `stop`, `next`, z', x

next : `main`, a, e, `iter`, z, `stop`, `next`, z'

Visibilité-suite

- p est le père de y si y est déclaré dans p.
- p est un ancêtre de y si p est soit y soit le père d'un ancêtre de y.
- Deux identificateurs sont frères s'ils ont le même père.

Si le corps d'une procédure p mentionne un identificateur i alors i est :

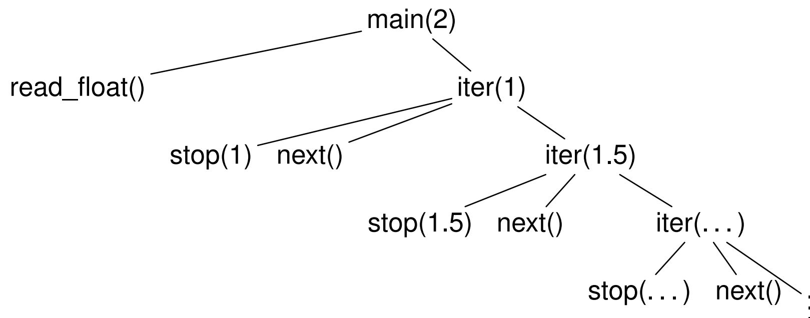
- soit déclaré localement dans p
- soit un ancêtre de p (par exemple p lui-même)
- soit un frère d'un ancêtre de p

Arbre d'activation

- Représente les exécutions possibles d'un programme.
- Les nœuds de cet arbre représentent des appels de procédures $p(e_1, \ldots, e_n)$.
- Un nœud a k fils $q_1, \ldots q_k$ si l'exécution du corps de la procédure donne lieu directement aux appels de procédures $q_1, \ldots q_k$ (ces appels pouvant eux-mêmes engendrer de nouveaux appels de procédures).
- La racine de l'arbre est l'exécution du programme principal.

Exemple

Sur notre exemple, on a :



Suite d’appels

- Lors d’une exécution du code on dira qu’une procédure est active si on n’a pas encore fini d’exécuter le corps de cette procédure.
- L’ordre des appels des procédures ne correspond pas à l’ordre de déclaration.
- Le long d’une branche de l’arbre, tous les tableaux d’activations des procédures actives sont empilées.

Propriété

- Lorsque qu'une procédure p est active alors tous les ancêtres de p sont des procédures actives.
- Récurrence sur la profondeur de l'appel de p dans l'arbre d'activation.
- Cas de base, le programme principal n'a que lui comme ancêtre.
- Soit une procédure active p, elle a été activée par une procédure q qui est toujours active.
  - Par hypothèse de récurrence les ancêtres de q sont actifs.
  - Pour des raisons de visibilité :
    - soit p est déclaré dans q
    - soit p est le frère d'un ancêtre de q

  Dans les deux cas, les ancêtres de p sont bien actifs

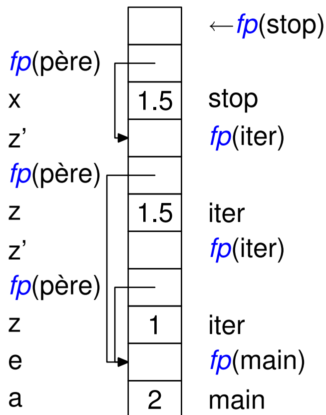
Accès aux variables

- Toutes les procédures actives ont leur tableau d'activation réservé en mémoire qui contient leurs variables locales.
- Si une procédure active mentionne une variable y celle-ci a été déclarée localement par un de ses ancêtres (peut-être lui-même).
- Le degré de parenté de cet ancêtre est connu à la compilation (différence de niveaux).
- En chainant chaque tableau d'activation d'une procédure p avec le tableau d'activation de son père statique, on retrouve simplement en suivant le bon nombre d'indirections l'endroit où la variable concernée est stockée.

Chaînage des tableaux d'activations

- On doit garder dans le tableau d'activation d'une procédure, l'adresse du tableau d'activation de son père.
- Cette adresse peut être stockée après le calcul des paramètres effectifs, avant l'appel : à l'endroit de *fp*-1.

Exemple



Calcul de l'adresse du tableau d'activation père

- Lorsque p appelle q, on calcule $n = \texttt{niveau}(p) - \texttt{niveau}(q)$.
  On a $n \geq -1$
- Le calcul de l'adresse du père de q se fait par la suite d'instructions :

$$\textbf{PUSHFP} \mid \underbrace{\textbf{LOAD} \ -1}_{n+1 \text{ fois}}$$

- On peut remarquer que la suite d'instructions **PUSHFP** | **LOAD** n est équivalente à **PUSHL** n.
- Lorsqu'on a mis à jour ce registre on peut lancer l'appel à la commande **CALL** q.
- L'appelant doit dépiler aussi cette valeur.

Calcul de l'accès à une variable

- Si la procédure p accède à une variable x, on calcule $n = \texttt{niveau}(p) - \texttt{niveau}(x)$.
- On a : $n \geq -1$
- On connait le décalage de x dans son tableau d'activation.
- Le code de l'accès à une variable dépend des niveaux relatifs de la procédure et de la variable.

$$code(x) = \textbf{PUSHFP} \mid \underbrace{\textbf{LOAD}\ -1}_{n+1 \text{ fois}} \mid \textbf{LOAD}\ \texttt{decal}(x)$$

Sommaire

1. Introduction
2. Code intermédiaire
3. Compilation des appels de fonctions
4. Compilation d'un langage fonctionnel
   - Langages à la Pascal
   - Fonctions comme objets de première classe
5. Compilation d'un langage objet

Rappels

Langages à la Pascal:

- Une fonction/procédure peut utiliser:
  - des variables globales
  - ses paramètres, ses variables locales
  - les paramètres et déclarations locales des fonctions/procédures dans lesquelles elle est définie.
- A l'appel, les tableaux d'activation des procédures parentes se trouvent dans la pile et permettent l'accès aux variables non locales.
  - On connait statiquement le lien de parenté
  - On chaine dynamiquement les tableaux d'activation
- Attention père statique de p (procédure dans laquelle p est définie) $\neq$ père dynamique (procédure qui appelle p)

Passage d'une fonction en paramètre

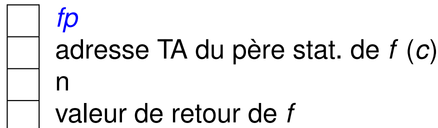
Pour compiler une procédure qui prend une fonction en paramètre, il faut deux informations:

- l'adresse du code de la fonction
- l'adresse du tableau d'activation du père de la procédure

```
let main () =
let b (h:int -> int) = print (h 2)
let c () =
     let m = ref 0 in
     let f n = !m + n
     in   b f
in c ()
```

Tableau d'activation de *c*:

Tableau d'activation de *f*:



Code de *f*:

PUSHL -1 | **LOAD** 0 | **PUSHL** -2 | **ADD** | **STOREL** -3 | **RETURN**

Exemple (2)

Tableau d'activation de *b*:

fp
TA du père stat. de *b* (*main*)
adresse du code de *h*
TA du père stat. de *h* (connu à l'appel)

Code de *b* (`print` (*h* 2)):

PUSHN 0 | **PUSHI** 2 | **PUSHL** -3 | **PUSHL** -2 | **CALL** | **POP** 2 | **CALL** `print` | **RETURN**

Exemple (3)

Lorsque *c* doit exécuter *b f*:

- il calcule le lien d'activation du père statique de *f* (ici *c*) comme s'il l'appelait et l'empile
- il empile l'adresse du code de *f*
- il empile l'adresse du TA de *b* (*main* qui est aussi le TA du père de *c*)

code(b f) =
PUSHFP | **PUSHA** f | **PUSHL** -1 | **CALL** b

Sommaire

1. Introduction
2. Code intermédiaire
3. Compilation des appels de fonctions
4. Compilation d'un langage fonctionnel
   - Langages à la Pascal
   - Fonctions comme objets de première classe
5. Compilation d'un langage objet

Langages fonctionnesl

Les fonctions sont des objets de première classe:

- peuvent être passées en argument
- peuvent être retournées comme valeur.

Renvoyer une fonction comme valeur

Si la fonction *f* calcule une fonction *g*

- le corps de *g* peut utiliser les variables de *f*.
- l'exécution du corps de *g* peut utiliser les variables de *f*.
- le TA de *f* disparait après la définition de *g*.

Il faut sauvegarder ces variables.

Exemple

```
let f x = let g y = x+y in g
let h = f 3
let j = f 4
h 5 + j 7
```

- g utilise x et y
- h est la fonction g avec $x = 3$
- j est la fonction g avec $x = 4$
- les appels à h et j instancient y

Clotures

- Une fonction $f\ x = e$ est formée
  - code pour exécuter le corps e
  - accès aux variables utilisées dans e
- on choisit de stocker les variables libres de e autres que x dans un environnement linéaire
- on appelle cloture l'association du code et de l'environnement
- l'environnement est alloué dans le tas
- statiquement chaque variable utilisée dans le corps de f est associée à un décalage dans l'environnement

Exemple

```
let f x = let g y = x+y in g
```

fonction	argument	environnement	corps
f:	x	[]	g
g:	y	[x]	x + y

Il faut déterminer un schéma dans lequel s'éxecute le corps de la fonction

fp
pointeur sur l'environnement
argument
valeur de retour

Valeur fonctionnelle

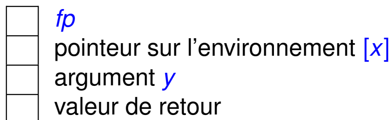
- Une valeur de type fonctionnel est formée de l'adresse d'un environnement et de l'adresse du code (alloués dans le tas ou sur la pile).

label code	
pointeur env	

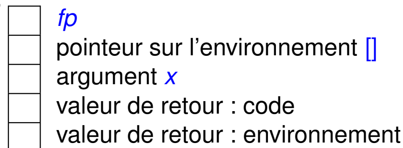
- On crée du code lorsqu'il y a le mot clé **fun** ($f\ x = e$ est équivalent à $f =$ **fun** $x \rightarrow e$).
- $(f\ 3)$ est une fonction : code de g, environnement $x = 3$.

Exemple

- Compilation du corps de *g*: $(x + y)$
  - TA de *g*:



  - code:
    PUSHL -1 | **LOAD** 0 | **PUSHL** -2 | **ADD** | **STOREL** -3 | **RETURN**
- Compilation du corps de *f*: (g)
  - TA de *f* :



  - code:
    ALLOC 1 | **DUP** | **STOREL** -4 | **PUSHL** -2 | **STORE** 0 | **PUSHA** g | **STROREL** -3 | **RETURN**

Appel

```
let h = f 3 (type int -> int)
let x = h 5 (type int)
```

- *h* est une variable dont la valeur est une fonction qui occupe une place double (environnement + code).
- *code*(let h = f 3) =
  PUSHN 2 | **PUSHI** 3 | **PUSHG** (dec(f)) | **PUSHG** (dec(f)+1) | **CALL** | **POP** 2 | **STOREG** (dec(h)+1) | **STOREG** (dec(h))
- *code*(h 5) =
  PUSHN 1 | **PUSHI** 5 | **PUSHG** (dec(h)) | **PUSHG** (dec(h)+1) | **CALL** | **POP** 2

Autre exemple

```
let twice h = let t x = h (h x) in t
let h = f 3 in twice h
```

fonction	argument	environnement	corps	type
twice:	h	[]	t	$(\alpha \to \alpha) \to \alpha \to \alpha$
t:	x	[h]	h (h x)	$\alpha \to \alpha$

Compilation du corps de *t*:

- TA de *t* :
  - *fp*
  - pointeur sur l'environnement [*h*]
  - argument *x*
  - valeur de retour
- code:
  PUSHN 1 | **PUSHL** -2 | **PUSHL** -1 | **LOAD** 0 |
  | **PUSHL** -1 | **LOAD** 1 | **CALL** | **POP** 2 | **PUSHL** -1 | **LOAD** 0
  | **PUSHL** -1 | **LOAD** 1 | **CALL** | **POP** 2 | **STOREL** -3 | **RETURN**

Schéma de compilation général

Compilation de **fun** $x \to e$

- On associe un label de code unique f pour cette fonction et on calcule l'ensemble $x_1, \dots, x_n$ des variables utilisées dans e qui ne sont ni x, ni locales dans e.
- On engendre pour f le code de e avec pour le code de x_i : **PUSHL** -1 | **LOAD** dec(x_i) (si x_i de taille 1)
- On crée l'environnement associé à $x_1, \dots, x_n$:
  ALLOC p – (taille totale nécéssaire pour $x_1, \dots, x_n$)
  | **DUP** | *code*(x_1) | **STORE** 0
  ... | **DUP** | *code*(x_n) | **STORE** dec(x_n)
- On renvoie le couple formé de l'environnement et de l'adresse du code.

Remarques sur la compilation des fonctions

- Dans le cadre d'un langage polymorphe, une valeur fonctionnelle doit aussi avoir une taille unitaire:
  - allocation dans le tas d'une paire fomée de l'environnement et de l'adresse du code.
- Considérer les fonctions comme unaires est inefficace (création de clôtures à chaque argument)
  - Compilation d'un code spécifique si la fonction est appliquée à tous ses arguments.

Remarques sur la génération de code

Des hypothèses qui simplifient le traitement :

- Valeurs de taille unitaire (essentiel si polymorphisme)
- Tableaux indicés à partir de zéro
- Vision uniforme des procédures et fonctions (juste la taille de la valeur de retour)
- Variables uniquement locales ou globales

Sommaire

1. Introduction
2. Code intermédiaire
3. Compilation des appels de fonctions
4. Compilation d'un langage fonctionnel
5. Compilation d'un langage objet
   - Représentation des classes et des objets
   - Héritage multiple
   - Appartenance à une classe

Introduction

- Les langages objets sont très populaires car ils offrent des mécanismes de structuration et d'encapsulation des données apropriés au développement de larges applications.
- Les mécanismes sophistiqués des langages objets doivent être pris en compte à la compilation.

Sommaire

1. Introduction
2. Code intermédiaire
3. Compilation des appels de fonctions
4. Compilation d'un langage fonctionnel
5. Compilation d'un langage objet
   - Représentation des classes et des objets
   - Héritage multiple
   - Appartenance à une classe

Compilation des objets

La déclaration d'une classe comporte

- des variables d'états
- des méthodes

Principes de la compilation:

- Les variables et méthodes statiques se comportent comme des variables globales et des procédures ordinaires.
- Un objet correspond à un bloc mémoire comportant les différentes variables d'état.
  Cette zone est allouée au moment de la création de l'objet.
- Les méthodes se comportent comme des procédures : elles ont des arguments déclarés et éventuellement un résultat.
  Elles s'appliquent à l'objet lui même (référencé par `this`).

Visibilité

- Les règles de visibilité des langages objet restreignent l'accès aux données.
- Les variables d'état privées introduites dans une classe ne seront visibles que dans les définitions de sous-classes.
- Les méthodes déclarées privées ne peuvent être utilisées en dehors de la classe.

Typage

- Les classes des langages objets jouent le rôle des types.
- Au centre des langages objets, se trouve la notion d'héritage.
- Une classe C peut hériter d'une classe D.
  - Les variables d'état de D sont variables d'état de C
  - les méthodes de D peuvent s'appliquer aux objets de la classe C.
  - Une méthode de D peut également être redéfinie dans C.
    Il y a alors surcharge du nom de la méthode : le même nom étant associé à plusieurs codes.
    Cette ambiguïté devra être résolue soit à la compilation soit à l'exécution.

Typage statique ou dynamique

Il y a deux manières de désambiguer le choix d'une méthode à appliquer :

- On détermine au typage la classe de l'objet et ceci permet de connaître statiquement la méthode à appliquer.
- C'est la classe de l'objet au moment de l'exécution qui détermine dynamiquement la méthode à sélectionner.

Typage dynamique

- Les classes des objets jouent un rôle important dans la compilation, aussi les objets manipulés comportent explicitement un pointeur vers le descripteur de leur classe.
- Le descripteur de classe comporte les informations utiles sur la classe, comme les labels associés aux méthodes définies dans la classe, les classes ancêtres, la taille des variables d'état ...

Héritage simple

- Dans le cas de l'héritage simple, une classe C hérite au plus d'une autre classe D.
- Organisation simple des variables d'états des objets de la classe C : extension des variables de la classe D
- Les méthodes compilées pour les objets de la classe D peuvent s'appliquer aux objets de la classe C.
- Un objet de la classe C qui hérite de D contient d'abord les variables de D puis celles de C.

Héritage simple : exemple

```
class A extends Object {var a:=0}
class B extends A {var b:=0 var c:=0}
class C extends A {var d:=0 }
class D extends B {var e:=0}
```

A:

a	0

B:

a	0
b	0
c	0

C:

a	0
d	0

D:

a	0
b	0
c	0
e	0

Héritage simple : exemple

- méthodes statiques : compilées comme des appels à des procédures (label de la méthode connu à la compilation).
- méthodes dynamiques le descripteur de classe doit contenir l'adresse du code à appeler.

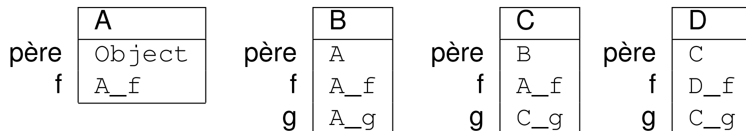
```
class A extends Object {var x:=0 method f()}
class B extends A {method g()}
class C extends B {method g()}
class D extends C {var y:=0 method f()}
```

Pour compiler `c.f()` il faut

- Trouver le descripteur de classe de *c*
- Chercher le label associé au code de *f*
- Appeler le code associé

Exemple-suite

- Le programme engendré comporte quatre fonctions correspondant au code déclaré dans les classes :
  labels `A_f`, `B_g`, `C_g` et `D_f`.
- Les descripteurs de classe sont stockés dans la pile à une adresse globale.
  Ils contiennent l'information sur les classes utiles à l'exécution :
  - l'adresse du descripteur de la classe père (pour implanter `instanceof`)
  - les labels des méthodes applicables



Exemple : objets

Un objet est représenté en mémoire à l'aide d'une ensemble de cellules :

- l'adresse du descripteur de la classe
- les valeurs des attributs dynamiques.

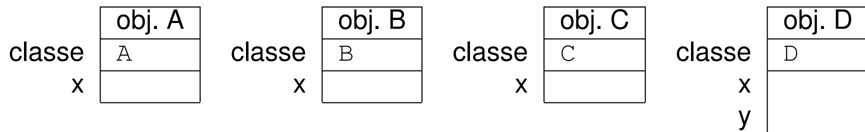
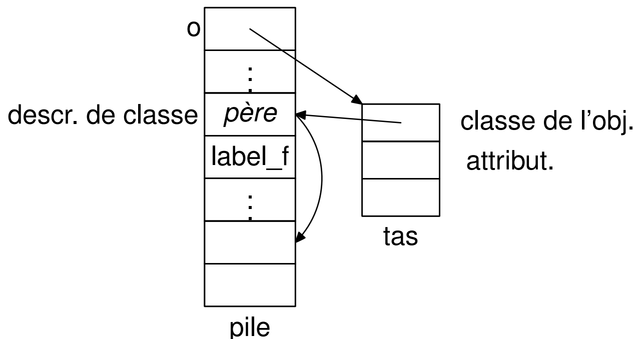


Table des symboles

- La table des symboles fournit une information statique, utilisée par le compilateur.
- Elle ne sert pas à l'exécution.
- Elle peut contenir pour chaque classe déclarée les informations suivantes :
  - adresse statique où sera stocké le descripteur de la classe,
  - classe père,
  - noms des attributs et décalages associés,
  - noms des méthodes applicables et décalage associé dans le descripteur de classe.
- La valeur d'un objet est l'adresse (dans le tas) de sa représentation. Pointe dans le bloc sur le descripteur de classe.

Compilation de l'appel de méthode

- On ne connaît plus statiquement l'adresse du code à appeler
- cette adresse se trouvera sur la pile



Compilation de l’appel de méthode

code(o.f(a)) =	**PUSHN** 1	valeur de retour
	code(o)	valeur de l’objet
	code(a)	valeur de l’argument
	code(o)	valeur de l’objet
	LOAD 0	adresse du descripteur de classe
	LOAD `decal`(f)	label de `f` dans le descr. de l’objet
	CALL	
	POP 2	

Sommaire

1. Introduction
2. Code intermédiaire
3. Compilation des appels de fonctions
4. Compilation d'un langage fonctionnel
5. Compilation d'un langage objet
   - Représentation des classes et des objets
   - Héritage multiple
   - Appartenance à une classe

Héritage multiple

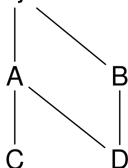
- Dans le cas de l'héritage multiple, une classe C peut hériter des objets des classes D_1 et D_2.
- On ne peut plus organiser simplement les variables de C, D_1 et D_2 pour que les procédures compilées pour D_i s'appliquent aux objets de C.

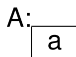
```
class A extends Object {var a:=0}
class B extends Object {var b:=0 var c:=0}
class C extends A {var d:=0}
class D extends A B {var e:=0}
```

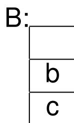
Décalages des attributs

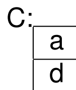
Trouver pour chaque variable d'état un décalage qui soit le même pour toutes les sous-classes.
Coloriage d'un graphe :

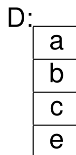
- sommet : variable d'état
- arête : variables qui coexistent dans une classe



A:


B:


C:


D:


Efficacité

- Si les objets étaient représentés en tenant compte de ces décalages alors il y aurait des trous dans la représentation.
- Seul le descripteur de classe a la forme adéquate, chaque place renvoie la position où la variable est stockée dans l'objet.
  - surcoût dans la manipulation des objets.
  - optimiser les accès au descripteur de classe.

Organisation des méthodes

- Pour organiser les méthodes il est également nécessaire de prendre en compte le graphe d'héritage et d'associer un décalage unique à chaque méthode.
- Ambiguïté possible sur le choix des méthodes.
- L'héritage multiple pose le problème de l'extensibilité (chargement dynamique de code).
- Il faut réorganiser le code ou bien être sûr d'associer un numéro unique.

Sommaire

1. Introduction
2. Code intermédiaire
3. Compilation des appels de fonctions
4. Compilation d'un langage fonctionnel
5. Compilation d'un langage objet
   - Représentation des classes et des objets
   - Héritage multiple
   - Appartenance à une classe

Tester l'appartenance à une classe

- Certains langages permettent de tester l'appartenance d'un objet à une classe (`instanceof`).
- Cela peut se faire en suivant les liens pères dans les descripteurs de classe (en supposant de l'héritage simple).
- On peut également garder dans chaque descripteur de classe un tableau des classes ancêtre.
  - Chaque classe a un niveau dans l'arbre d'héritage, `Object` de niveau 0.
  - Pour savoir si un objet x est une instance de la classe C de niveau j, il suffit de regarder la j-ème valeur dans le descripteur de x et vérifier que c'est C.
  - Certains langages autorisent à restreindre la classe d'un objet par un cast. Avec ou sans vérification (Java versus C++)

Optimisations

- Le coût le plus important dans un langage objet est l'appel de méthode.
- Il est essentiel d'analyser le code pour remplacer les appels dynamiques par des appels statiques.
- Par exemple, si une méthode n'est pas redéfinie dans une sous-classe alors il n'y a qu'un seul code possible à exécuter.