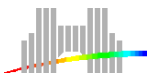
Organisation du programme objet

Paul Feautrier

ENS de Lyon
Paul.Feautrier@ens-lyon.fr
perso.ens-lyon.fr/paul.feautrier

2 février 2009



Introduction

Avant de commencer à générer le programme objet, il faut définir un modèle d'exécution, qui doit permettre l'implémentation de tous les aspects du langage source :

- Exemple : si le langage permet des tableaux extensibles (Matlab) ou des tables hashcodée (Ocaml, Tk/Tcl, Maple), le modèle doit comporter les outils nécessaires (gestion dynamique de la mémoire).
- Le modèle doit permettre une implémentation raisonablement efficace.
- Exemples de contraintes :
  - Récursivité
  - Gestion dynamique de la mémoire
  - Compilation séparée, bibliothèques, relations avec le système d'exploitation, entrée / sorties
  - Tableaux
  - Objets et méthodes

Contraintes, I : questions de temps

On doit distinguer le temps de la compilation et le temps de l'exécution.

- Le compilateur n'a pas accès directement à ce qui se passe à l'exécution.
- Inversement, les objets du compilateur ont disparu (en général) au moment de l'exécution.

  Par exemple, problème des tables de symboles au moment du deboguage, option -g du compilateur C.

- Pour agir à l'exécution, le compilateur doit insérer du code dans le programme objet.

Récursivité, règles de visibilité

- Dans presque tous les langages les programmes peuvent être structurés en procédures (fonctions, sous-programmes) qui peuvent être récursive.
- Il s'en suit que la gestion de la mémoire ne peut être statique – le compilateur ne peut pas calculer toutes les adresses.
- Tous les langages ont des règles de visibilité qui définissent des espaces de nommage indépendants (exceptions : langages interprétifs).
- Les règles de visibilité diffèrent d'un langage à l'autre (exemples : C et Pascal).

Gestion dynamique de la mémoire

Certains langages permettent la création dynamique de structures de données.

- Le programme objet doit dialoguer avec le système d'exploitation
- Si l'on veut une gestion automatique des données, le compilateur doit instrumenter les structures :
  - prévoir un compteur de références
  - marquer les pointeurs ou les identifier d'une autre façon

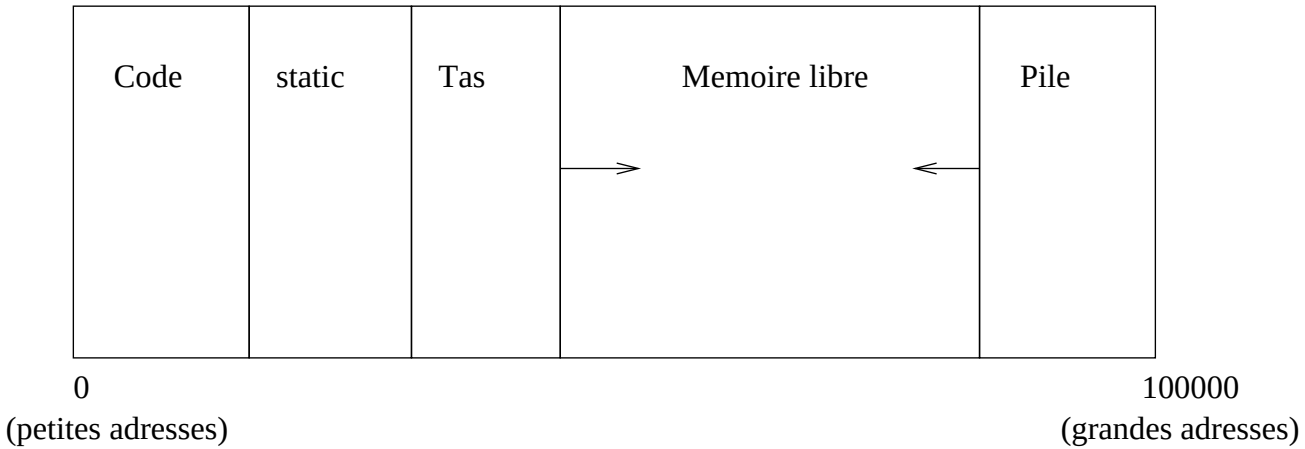
Compilation séparée

- Possibilité de découper un programme en modules pouvant être compilés indépendamment.
- En particulier, utilisation d'une bibliothèque
- Deux approches :
  - à la Fortran (exemple C) : il n'y a aucune communication entre modules. Pour maintenir un minimum de cohérence, on demande au programmeur de *recopier* certaines informations (les déclarations).
  - l'ordre des compilations est arbitraire
  - à la Ada (exemple Ocaml) : le compilateur enregistre une partie de la table des symboles après chaque compilation.
  - l'ordre des compilations est contraint

Entrées /sorties

- Certains langages n'ont pas d'instructions d'entrée sortie (C, Ocaml) ; elles sont déléguées à une bibliothèque.
- Inconvénient : les fonctions ne peuvent pas être génériques. Exemple du `close_out` en Ocaml.
- D'autres langages ont des instructions d'entrée sortie, qui doivent être compilées.

Organisation de la mémoire, I



- Ne pas confondre espace d'adressage et espace mémoire.
- Le compilateur suppose toujours que le programe qu'il construit est seul en mémoire.
- En général, c'est le dispositif de mémoire virtuelle qui permet cette hypothèse.
- La disposition ci-dessus est la plus courante, mais les conventions peuvent changer suivant le processeur (certains processeurs réservent la moitié de l'espace d'adressage pour le système).

Organisation de la mémoire, II

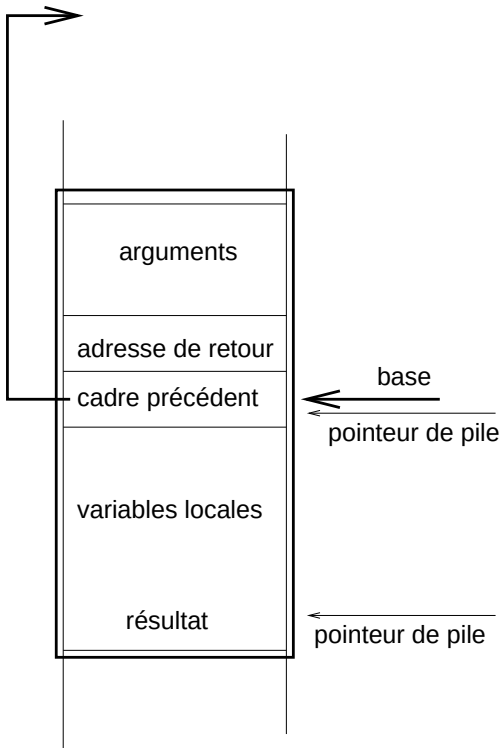
Si le processeur n'a pas de mémoire virtuelle (systèmes embarqués, microcontroleur).

- **Code relatif** : on spécialise un registre pour contenir l'adresse de base du code, et ce registre est additionné à toutes les adresses. Même méthode pour les données.
- Pour déplacer le code, il suffit de changer la valeur du registre de base.
- **Code translatable** : On écrit le code comme si l'adresse de base était 0.
- Au chargement, on additionne l'adresse de base à toutes les adresses.
- Il faut prévoir une zone hors texte pour savoir si une instruction contient une adresse ou non.

Pile d'exécution, principe

- Chaque fois qu'une procédure est activée, elle a besoin de place en mémoire pour ses variables locales et ses arguments.
- Si une procédure est récursive, il peut y avoir plusieurs activations en cours à un instant donné.
- L'espace nécessaire doit donc être alloué dynamiquement au moment de l'appel, et libéré au moment du retour.
- Mais on peut faire mieux que d'utiliser un outil comme `malloc` parce que les entrées et sorties de procédures se font dans l'ordre Last In First Out.
- On peut donc utiliser une *pile*.

Pile d'exécution, structure



- La pile est divisée en *cadres* ou enregistrements d'activation.
- Chaque cadre comporte :
  - Les arguments de la procédure
  - L'adresse de retour
  - Un lien vers le cadre de la procédure appelante
  - Les variables locales, et en particulier un emplacement pour le résultat de la procédure.
- Un register spécialisé, la base, pointe vers le lien descendant

Comment ça marche

- Adressage des variables locales : `*(base - 4)`
- Adressage des arguments : `*(base + 8)`
- Séquence d'appel et de retour :

```
  tos += sizeof(vars);  //tos :: Top of Stack
  *(++tos) = arg1;
  *(++tos) = arg2;
..
  call();
  tos -= sizeof(vars) + sizeof(args);
```

- Le compilateur déduit des déclarations la taille des variables et des arguments ; il calcule les déplacements par accumulation et les enregistre dans la table des symboles

Résultat de la fonction

- Si le processeur le permet, le résultat est retourné dans un registre.
- Si le résultat est trop gros, il peut être retourne sur la pile.
- Cas extrème : le résultat est rangé sur le tas, et un pointeur est retourné dans un registre.

Appel par valeur, appel par référence

- En C tous les paramètres sont transmis par copie : la procédure appelée dispose d'une copie du paramètre qui fonctionne comme une variable locale. Les modifications ne sont pas répercutées.
- Dans certains langages (Pascal, C++), un paramètre peut être transmis par référence. Equivalent à la transmission d'un pointeur en C.
- Mais c'est le compilateur qui doit déréférencer le pointeur, alors qu'en C c'est à la charge du programmeur.

Gestion des registres

La procédure appelante ne peut pas savoir ce que la procédure appelée va faire des registres du processeurs. Deux solutions :

- La procédure appelante sauvegarde les registres qu'elle juge utile avant l'appel
- La procédure appelée sauvegarde les registres qu'elle va utiliser avant de commencer
- la sauvegarde se fait sur la pile d'exécution
- certains processeurs ont des instructions de sauvegarde / restauration rapide des registres
- La décision dépend de plusieurs facteur : politique d'allocation de registres, nombre de registres, etc.

Accessibilité

Suivant les langages, les règles d'accès aux objets étrangers varient.

- En C et Java, les procédures ne s'emboitent pas, et les objets sont soit locaux (accés relatif à l'enregistrement d'activtion courant), soit globaux (adressage absolu). Voir cependant les extensions sauvages du compilateur gcc.
- En Pascal, ADA, les procedures s'emboitent, et il est possible d'accéder aux objets des procédures englobantes, si toute fois ils ne sont pas masqués par un objet local.
- En particulier, il n'y a qu'un seul enregistrement accessible par procédure récursive, le plus récent.
- On peut accéder aux objets non locaux en déroulant la chaîne des pointeurs d'activation, mais c'est inefficace.

Méthode du *display*

- Chaque procédure a une profondeur lexicale (sa profondeur dans l'arbre de syntaxe abstraite.
- A chaque instant, il n'y a qu'une seul procédure visible à chaque profondeur lexicale.
- On peut donc préparer un tableau, le *display*, qui contient pour chaque profondeur lexicale un pointeur vers l'enregistrement d'activation de la procédure visible.
- Ce tableau doit être entretenu à chaque entrée dans une procédure et à chaque retour s'il y a eu possibilité de récursivité.

Relations avec le système d'exploitation

Si le matériel le permet, elles sont réduites au minimum – en fait, camouflées dans des fonctions de bibliothèque.

- Extension de la pile : transparente, sur détection d'une violation de la protection mémoire.
- Appels au système (en particulier entrées/sorties, terminaison et gestion du tas), implémentées sous forme d'une bibliothèque
- Lancement d'un programme : le système fait un saut vers une adresse fixe, où l'éditeur de lien a placé un programme qui simule un appel de la fonction `main` (en C).

Gestion dynamique de la mémoire

Indispensable dans le langages modernes. Deux classes de méthodes :

- La mémoire est gérée par le programmeur (C, C++).
- Avantage : pas de contrainte sur les structures de données et la manipulation des pointeurs.
- Inconvénient : c'est difficile à écrire, et il est presque impossible de ne pas faire d'erreurs (*fuites de mémoire*).
- La mémoire est gérée automatiquement par le système d'exécution (Java, Lisp, Ocaml)
- Avantage : facile et sûr
- Inconvenient : contraintes sur l'utilisation des pointeurs
- Inconvénient : le système de gestion peut dégrader les performances

Gestion des tableaux, I

La gestion des tableaux dépend fortement du langage.

- **Tableaux de taille fixe** (Fortran, C ANSI)
  - Le tableau est alloué soit dans l'espace des données statiques, soit dans l'enregistrement d'activation.
  - Le compilateur connaît son adresse ou son déplacement par rapport à la base.
- **Tableaux de taille variable** (Pascal, addition récente à gcc)
  - Concerne seulement les arguments de procédure.
  - La taille du tableau doit pouvoir être calculée à l'entrée de la procédure.
  - Le compilateur doit générer du code pour calculer la taille du tableau et son déplacement dans l'enregistrement d'activation

Gestion des tableaux, II

Certains compilateurs associent à chaque tableau un *descripteur* de taille fixe (dépendant du nombre de dimension).

- Le descripteur peut être alloué sur la pile
- Le tableau proprement dit est alloué sur le tas
- Le descripteur contient un pointeur sur le tableau et la liste des dimensions.
- **Avantages** Le descripteur regroupe tout ce qu'il faut savoir sur le tableau. Il est facile de programmer des opérations sur tableaux (langages à la Fortran 95)
- **Inconvenient** Lourdeur

Gestion des tableaux, III

Dans tous les cas, le compilateur doit engendrer le code de calcul d'adresse :

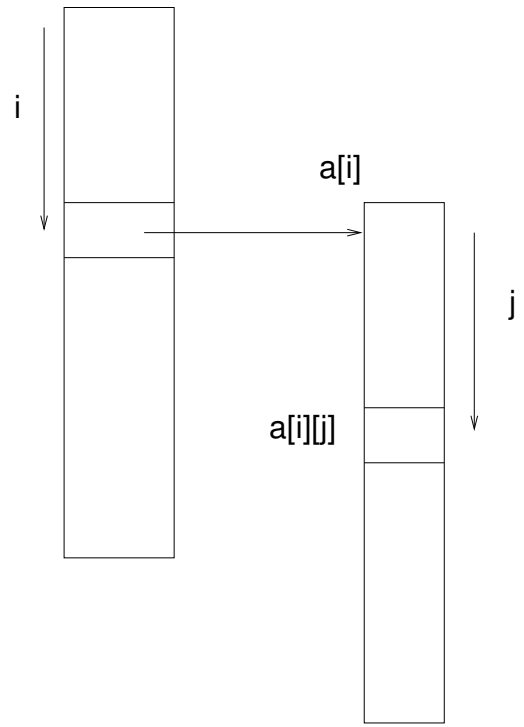
```
float a[n1][n2][n3];

      a[i][j][k]
```

$$\&\texttt{a[i][j][k]} = \&a + (n_1 i + j)n_2 + k$$

De plus, dans certains langages (Pascal), le compilateur doit engendrer un code de vérification ($0 \leq i < n_1$, etc.)

Gestion des tableaux, IV



- En Java, il n'y a que des tableaux à une dimension.
- Un tableau à plusieurs dimensions contient des pointeurs sur des tableaux de dimension inférieure.
- Tous les tableaux sont alloués sur le tas

Gestion des tableaux, V

Le calcul d'adresse se fait par indexations successives :

```
float a[n1][n2][n3];
```

```
a[i][j][k]
```

$$
\begin{aligned}
\&\texttt{a[i]} &= \&a + i, \\
\&\texttt{a[i][j]} &= \texttt{a[i]} + j, \\
\&\texttt{a[i][j][k]} &= \texttt{a[i][j]} + k
\end{aligned}
$$

Le compilateur doit également tester les indices.
Inconvénient lourdeur : multiples accés mémoire

Chaînes de caractères

Dans certains langages (C) les chaînes de caractères sont des tableaux ; dans d'autres (Ocaml, Pascal), elles constituent un type de base.

- Particularité : taille variable.
- Solution : surdimensionnement.
- Problème : la représentation de la longueur effective.
- Pascal : le premier caractère code la longueur, longueur limitée à 255 caractères.
- C : marqueur de fin.

En général, le compilateur délègue les opérations sur chaînes de caractères à une bibliothèque.

Représentation des booléens

- La convention usuelle est 0 = faux, 1 = vrai.
- Un booléen peut être logé dans un octet.
- Certains langages traitent les *vecteurs de bits* (C, les langages de description du matériel).
- Tous les processeurs ont des opérations de manipulation des bits et vecteurs de bits.
- On peut compiler les expressions booléennes comme les expressions arithmétiques (voir plus lion).
- Mais très souvent les booléens ne sont que des intermédiaires qui ne sont pas affectés à une variable.

Comparaisons

- Les processeurs possèdent en général un registre de *flags* qui enregistre des informations sur le résultat de la dernière opération arithmétique.
- On peut ensuite exécuter un branchement conditionné par ces flags et en déduire un booléen.

Tests, I

En présence d'un test portant sur une expression booléenne complexe :

- Evaluer les comparaisons
- Calculer l'expression booléenne
- Tester le booléen résultant

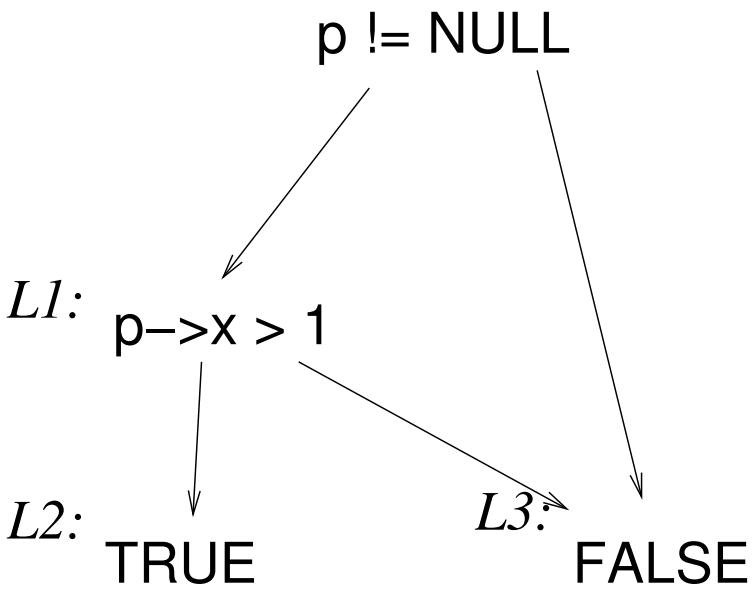
Tests, II

Autre méthode :

- On construit un arbre de décision à partir de l'expression booléenne
- On code l'arbre de décision sous forme d'une cascade de branchements conditionnels
- La qualité du résultat dépend fortement de l'odre du codage.
- La méthode implémente naturellement *l'évaluation paresseuse*

Exemple

```
if(p != NULL && p->x > 1) S;
```




```
    if(p == NULL) goto L3;
    if(p->x <= 1) goto L3;
L2 : S;
L3 :;
```

Boucles

Deux sortes de boucles :

- Les boucles `while` qui se compilent comme des instructions de contrôle.
- Les boucles `DO`, pour lesquelles on peut identifier un compteur et prédire le nombre de tours.
  - Pascal et Fortran : les boucles `DO` ou `for` ont une syntaxe spéciale
  - En C, il n'y a aucune garantie qu'une boucle `for` est une boucle `DO`. Il faut analyser (par pattern matching) l'initiateur, le test et la progression, et vérifier que le compte tour n'est pas modifié dans la boucle.
  - On peut aller plus loin et rechercher les boucles `while` qui sont des boucles `DO` déguisées.
- Une boucle `DO` s'optimise en allouant le compteur à un registre et en la transformant en une boucle vers 0.

Techniques de génération de code, I

Le compilateur doit engendrer le programme objet, c'est-à-dire composer et écrire des chaînes de caractères (langage cible, assembleur).

- Il est déconseillé de *cabler* ces chaînes dans le code (difficulté de localisation et de modification).
- Les fragments de code doivent être groupés et utilisés de façon uniforme. Exemple du compilateur `pcc` (Portable C Compiler).

```
#define ADD 0
#define LOAD 1
#define LOADI 2
char *assembly[10];
assembly[ADD] = "%s\t add\t R%d,R%d,R%d\n";
assembly[LOAD] = "%s\t load\t R%d,R%d,%s\n";
assembly[LOADI] = "%s\t loadi\t R%d,%d\n";
...
fprintf(out, assembly[LOAD], label, destination, base, variable)
```

Permet de changer facilement d'assembleur.

Génération de code, II

Techniques plus sophistiquées

- Utiliser des macros et le préprocesseur C `cpp`
- Faire le travail de génération au niveau de la RI, et écrire un *pretty printer*
- Si la RI est à objet, associer une méthode de génération à chaque classe (par exemple redéfinir la méthode `toString()` de Java).
- Ecrire une grammaire générative pour le langage objet et écrire un interpréteur : une production peut contenir :
  - un terminal
  - un non terminal
  - une demande d'information à la RI.

Donnez libre cours à votre ingéniosité.