Architecture des Ordinateurs
Partie III : Liens avec le système d'exploitation
2. Génération de code

David Simplot
simplot@fil.univ-lille1.fr

Objectifs

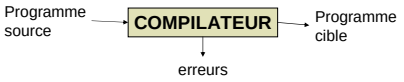
- Voir la génération de code ainsi que les mécanismes implicites utilisés
  - Gestion des données
  - Gestion des appels de fonctions
  - Gestion de l'allocatio dynamique

Au sommaire...

- **Schéma général d'un compilateur**
- Organisation de l'espace mémoire
- Accès aux noms non locaux
- Passage de paramètres
- Techniques pour l'allocation dynamique

Schéma général d'un compilateur (1/8)

- Vision simplifiée d'un compilateur :



- Programme source
  - C, C++, Java, ADA, Cobol, Fortran, Pascal…
- Programme cible
  - Programme « binaire » en langage machine
  - Bytecode (type bytecode Java ou P-Code MS)

Schéma général d'un compilateur (2/8)

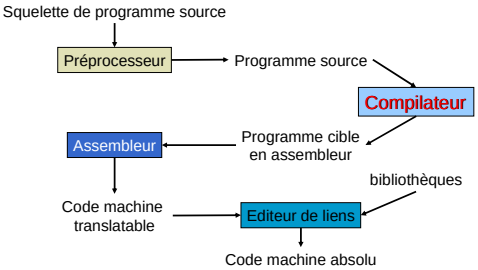


Schéma général d'un compilateur (3/8)

- Modèle de la compilation par analyse et synthèse
  - Analyse = partitionner le programme et construire une représentation intermédiaire des parties
    - Voir cours d'ASC
      - Analyse lexicale = identifier les tokens
      - Analyse syntaxique = déduire les structures → structure d'arbres abstraits
      - Analyse sémantique = contrôle des types, bon utilisage des opérateurs, etc.
  - Synthèse = contruire le programme cible désiré à partir de la représentation intermédiaire

Schéma général d'un compilateur (4/8)

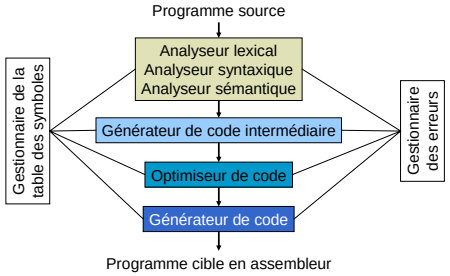


Schéma général d'un compilateur (5/8)

- Rôle de la table des symboles
  - Stocker les identificateurs utilisés dans le programme
    - Déterminer leurs types (variables, fonctions, etc…)
      - Pour les fonctions : le nombre et le type des arguments
    - Déterminer leurs portées dans le programme
- Le code intermédiaire est généré à l'issue de la phase d'analyse sémantique

Schéma général d'un compilateur (6/8) Exemple

- Programme source
  - note = vrai_note * 2 + mini
- Analyseur lexical
  - id1 ← id2 * entier + id3
- Analyseur syntaxique

Table des symboles

1	note	float
2	vrai_note	float
3	mini	float

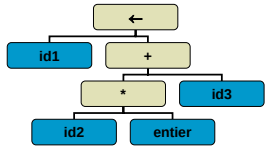


Schéma général d'un compilateur (7/8) Exemple (suite)

- Analyse sémantique

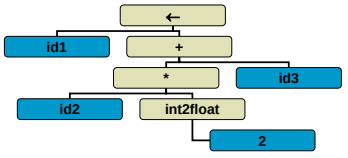


Schéma général d'un compilateur (8/8) Exemple (suite)

- Générateur de code intermédiaire

```
tmp1 ← int2float(2)
tmp2 ← id2 * tmp1
tmp3 ← tmp2 + id3
id1 ← tmp3
```

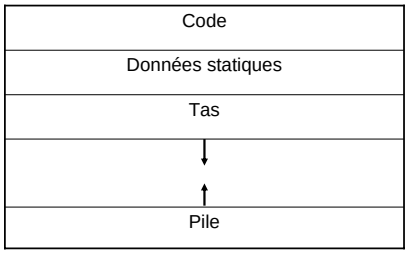
- Optimiseur de code
  - `tmp1 ← id2 * 2.0`
  - `id1 ← tmp1 + id3`
- Générateur de code

```
MOVF R1, id2
MULF R1, %2.0
MOVF R2, id3
ADDF R1, R2
MOVF id1, R1
```


Au sommaire...

- Schéma général d'un compilateur
- **Organisation de l'espace mémoire**
- Accès aux noms non locaux
- Passage de paramètres
- Techniques pour l'allocation dynamique

Organisation de l'espace mémoire (1/4)

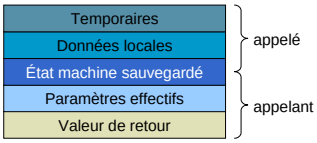


Organisation de l'espace mémoire (2/4)

- Tas
  - Sert pour l'allocation dynamique de mémoire
    - *E.g.* le malloc
- Pile
  - Sert pour l'« appel des fonctions »
    - Idem pour les invocations de méthodes en langage objet
  - À chaque appel de fonction, on se sert de cette pile pour mettre un « enregistrement d'activation »
    - Aussi appelé « frame »

Organisation de l'espace mémoire (3/4)

- Exemple d'enregistrement d'activation :



  - État machine sauvegardé
    - Appelant : PC lors de l'appel à la sous-routine (CALL/JSR)
    - Appelé : sauvegarde des registres

Organisation de l'espace mémoire (4/4)

- La taille de chacun des champs « données »
  - Aussi bien données locales, paramètres que valeur de retour
- dépend de la représentation en mémoire des types du langage de haut-niveau
- L'accès aux variables locales et au paramètres passent donc par la pile (par le frame).
- Problème :
  - La taille du programme est limité ⇒ on majore la taille du tas et de la pile ⇒ gaspillage d'espace mémoire dans la plupart des cas
  - Solution : chaînage de pile et de tas…

Au sommaire...

- Schéma général d'un compilateur
- Organisation de l'espace mémoire
- **Accès aux noms non locaux**
- Passage de paramètres
- Techniques pour l'allocation dynamique

Accès aux noms non locaux (1/3)

- Variables globales ou variables de classes
  - Adresse connue à la compilation ou on a un « adresseur » qui connaît l'adresse de réification de la classe
- Variables de bloc

```
int main()
{
   int a=0;
   int b=3;
   {
     float b=1.5;
     …
   }
}
```

Bloc 1 (float b=1.5; …) ; Bloc 0 (int a=0; int b=3; Bloc 1 }).

Accès aux noms non locaux (2/3)

- On renomme les variables

```
int main()          int main()
{                   {
  int a=0;            int a=0;
  int b=3;            int b0=3;
  {                   {
    float b=1.5;        float b1=1.5;
    …                   …
  }                   }
}                   }
```


Accès aux noms non locaux (3/3)

- C'est à la charge de l'optimiseur de déterminer la durée de vie des variables et de réutiliser l'espace mémoire si possible

Au sommaire...

- Schéma général d'un compilateur
- Organisation de l'espace mémoire
- Accès aux noms non locaux
- **Passage de paramètres**
- Techniques pour l'allocation dynamique

Passage de paramètres (1/6)

- Passage par valeur
  - Ce qui est mis dans le frame, c'est une copie de la valeur de l'argument
- Passage par référence
  - On met dans le frame l'adresse de l'argument
  - Il s'agit d'un pointeur
  - Cette adresse pointe sur une données accessible par la fonction appelante
- Passage par donnée-résultat
  - Mixe des deux précédents
  - On donne un pointeur et un peut éventuellement utiliser la valeur stockée à l'adresse passée en paramètre

Passage de paramètres (2/6) Exemple

```
void swap(int *a, int *b)
{
    int tmp;
    tmp = *a;
    *a = *b
    *b = *a;
}

void tri(int *a, int *b)
{
    if ( *a > *b) swap(a, b)
}
```

```
int min(int a, int b)
{
    if ( a<b )
          return a;
    return b;
}

main()
{
    int x=7, y=5, z;
    z=min(x, 3);
    tri(&x, &y);
}
```


Passage de paramètres (3/6) Exemple (suite)

```
Function main
  .return none
  .param
  .locals
     x int
     y int
     z int
  .temp
     tmp1 db 4 dup(?)
  .code
     x ← 7
     y ← 5
     sp ← sp + 4 ; réservation
                 ; retour
```

```
push x                ; arg1
     tmp1 ← 3
     push tmp1        ; arg2
call min
sp ← sp - 8
pop tmp1        ; retour
z ← tmp1
tmp1 ← @x
push tmp1       ; arg1
tmp1 ← @y
push tmp1       ; arg2
call swap
end function    ; fin main
```

Passage de paramètres (4/6) Exemple (suite)

- Function min
  .return int
  .param
  a int
  b int
  .locals
  .temp
  tmp1 int
  tmp2 int
  .code
  tmp1 ← a
  tmp2 ← b

```
    cmp tmp1, tmp2
     jb min_suite
     return ← tmp1   ; retour
    jmp min_fin
min_suite:
    return ← tmp2   ; retour
min_fin:
    end function  ; fin min
```


Passage de paramètres (5/6) Exemple (suite)

- Appel de la fonction min

Passage de paramètres (6/6) Exemple (suite)

- Function swap
  .return none
  .param
  a int*
  b int*
  .locals
  tmp int
  .temp
  tmp1 int

```
.code
   tmp ← [a]
   tmp1 ← [b]
   [a] ← tmp1
   [b] ← tmp
End function
```

- Exécution de la fonction swap

Au sommaire...

- Schéma général d'un compilateur
- Organisation de l'espace mémoire
- Accès aux noms non locaux
- Passage de paramètres
- **Techniques pour l'allocation dynamique**

Techniques pour l'allocation dynamique

- But : construire des données/objets persistant à la destruction du contexte d'une fonction
- Il s'agit de « gérer » le tas
- Implémentation « bitmap »
- Implémentation par « chaînage » des emplacements libres
  - On regroupe par « slots » d'une taille fixée
  - On chaîne les emplacements vides

Conclusion

- Optimisation du code intermédiaire
  - Durée de vie des variables
- Génération de code natif
  - Encore une phase d'optimisation
    - Pour utiliser les instructions les moins coûteuses du microprocesseur
    - À chaque instruction, on peut attribuer un coût qui est le nombre de cycles nécessaire pour faire l'instructions
      - De plus en plus difficile avec les processeurs optimisants
  - Réécriture avec des règles de traductions des instructions du code intermédiaire
  - Cross-compilateur ?