Introduction à la Compilation

S. Salva
MdC Hdr ,Université d'auvergne, Limos

Introduction

- Un compilateur est un programme qui traduit un programme écrit dans un premier langage (programme source) et le traduit en un programme équivalent écrit dans un autre langage (cible).

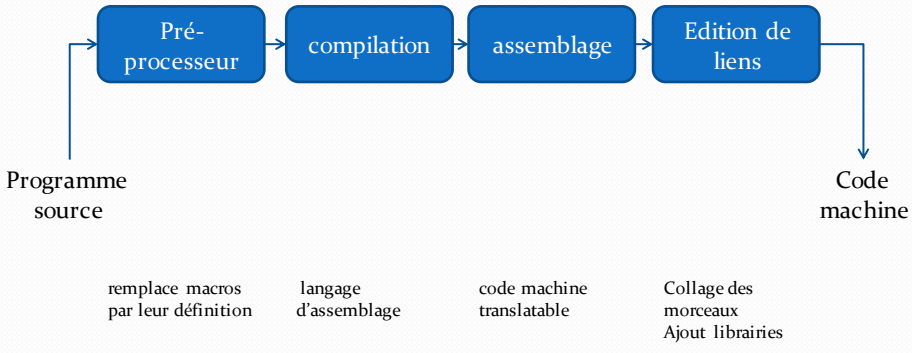
- La compilation fait appel à :
  - – la théorie des langages;
  - – l'architecture des machines;
  - – l'algorithmique et le génie logiciel.

Références

- A. W. Appel. Modern Compiler Implementation in ML. Cambridge, University Press, 1998
- J. R. Levine, T. Mason, and D. Brown. Lex & Yacc. Unix Programming Tools. O' Reilly, 1995
- Cours de compilation, Jacques Ferber, http://www.lirmm.fr/~ferber/Compilation/compil.htm

3

Introduction



4

Préprocesseur

- En C:
- #define dans le code source
- Commande cpp (C PreProcessor) :
  - cpp source.c (sortie sur stdout)
  - Cette commande est équivalente à l'option -E de gcc,
  - lors de la compilation, gcc appelle cpp implicitement.

5

Compilation séparée en langage C

1. **Utilité de la compilation séparée**

- La compilation séparée permet une conception et une réalisation plus rapide (analyse, écriture, compilation, mise au point, maintenance), par une structuration ou un découpage de l'application (approche modulaire).
- Le travail en équipe est ainsi facilité. La compilation séparée permet aussi la réutilisation du code et l'utilisation d'autres langages de programmation.

6

Compilation séparée en langage C

2. Contenu d'une unité de compilation (UC)

- Une unité de compilation correspond à un source .c
  Elle contient des :
  - instructions
  - directives (pour le préprocesseur et le compilateur)
  - commentaires
  - déclarations et définitions de variables et de fonctions :
- Au cours de l'exécution d'un programme, les identificateurs (noms des variables et des fonctions) sont remplacés par leurs adresses.

7

Compilation séparée en langage C

- Linker: programme qui rassemble les modules objets constituants une application pour fabriquer un autre module-objet éditable.
- Il réalise la correspondance entre référence externe et point d'entrée.
  Il génère soit un exécutable si toutes les références externes sont résolues, soit un .o s'il subsiste des références non définies.

Compilation statique :

- Par défaut, les bibliothèques sont liées statiquement, le code des fonctions des bibliothèques sont ajoutées dans l'exécutable.
- Les fichiers de bibliothèques ont pour extension .a (MS : .lib)

8

Compilation séparée en langage C

- **Compilation statique :**

Avantages :

- Exécutable autonome, contient tout le code nécessaire à son exécution
- Possibilité d'avoir des fonctions écrites dans d'autres langages de programmation

Inconvénients :

- Taille de l'exécutable plus grosse
- Code non factorisé

- **Compilation dynamique :**

Exécutable moins gros, économie de mémoire centrale, pas besoin de recompiler les applications qui utilisent la bibliothèque partagée si elle est modifiée.

9

Étude d'un appel au compilateur gcc

- gcc -o alui deux.c trois.c un.o tic.a -lld

- générer un module objet exécutable nommé alui à partir de deux unités de compilation (deux.c trois.c), d'un module objet éditable (un.o), d'une bibliothèque tic.a, d'une bibliothèque libld.a (nom déduit de l'option -l suivie de ld) et de la bibliothèque standard (non citée donc implicite) du langage c (libc.a).

- Rappel 1 : Il existe un répertoire standard où sont fournies des bibliothèques. Les noms de ces bibliothèques sont par convention de la forme libxxx.a.
  L'option -l suivie de xxx fait référence à ce répertoire standard et au fichier libxxx.a.

- Rappel 2 : Un module objet éditable est exécutable lorsque toutes ses références externes sont résolues.

10

Étude d'un appel au compilateur gcc

- L'appel du programme gcc entraîne l'exécution d'une série de programmes :

1) Appel du pré-processeur cpp puis du traducteur pour chacune des UC (Unité de compilation) deux.c, trois.c. Donc générer les modules objets éditables deux.o, trois.o

2) Appel de l'éditeur de liens ld pour lier entre eux un.o, deux.o et trois.o en un module objet éditable M1.o.

3) Si la TRE (table des références externes) de M1.o est vide alors M1.o est renommé alui et fin.
Sinon étape 4.

4) Résolution des références externes de M1.o à l'aide des modules objets éditables du fichier bibliothèque tic.a pour produire un un module objet éditable M2.o.

Étude d'un appel au compilateur gcc

5) Si la TRE (table des références externes) de M2.o est vide alors M2.o est renommé alui et fin. Sinon étape 6.

6) Résolution des références externes de M2.o à l'aide des modules objets éditables du fichier bibliothèque libld.a pour produire un module objet éditable M3.o.

7) Si la TRE (table des références externes) de M3.o est vide alors M3.o est renommé alui et fin. Sinon étape 8.

8) Résolution des références externes de M3.o à l'aide des modules objets éditables du fichier bibliothèque standard du langage c (libc.a) pour produire un module objet éditable M4..o.

9) Si la TRE (table des références externes) de M4.o est vide alors M4.o est renommé alui et fin.
Sinon message d'erreur de l'éditeur de liens : Références externes non résolues" et fin

Makefile

```
philo : philo.o logfile.o ezsem.o
        gcc -o philo philo.o logfile.o ezsem.o

philo.o : philo.c
    gcc -c philo.c -W –Wall

logfile.o : logfile.c
        gcc -c logfile.c -W –Wall

ezsem.o : ezsem.c
        gcc -c ezsem.c -W -Wall
```

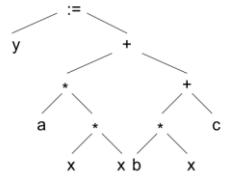
13

Compilation: 2 étapes

- Analyse et synthèse

- partie analyse : sépare les constituants du prog. source et produit une représentation intermédiaire
- partie synthèse : génère le prog. cible à partir de la rep. intermédiaire
- représentation intermédiaire : souvent un arbre abstrait. – noeuds : opérations du programme source – feuilles : arguments de ces opérations.

$y := ax^2 + bx + c$



14

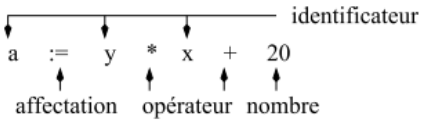
Analyse

- Exemple d'outils utilisant des techniques d'analyse :
  - – éditeur structurel : analyse de la structure, fournit les mots clés du langage, vérifie l'adéquation entre début et fin de bloc…
  - – interpréteurs de commandes.
- Analyse du programme source en 3 phases :
  - – analyse lexicale;
  - – analyse syntaxique;
  - – analyse sémantique.

15

Analyse lexicale ou analyse linéaire

- Le but de l'analyseur lexical est de reconnaître les unités lexicales ou lexèmes :
  - les identificateurs et les mots clés du langage;
  - l'affectation et les opérateurs (+,*,…).



16

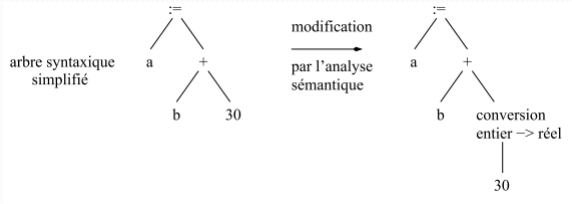
Analyse syntaxique

- On peut également définir la notion d'instruction :
  - Soit *id* un identificateur et *exp* un expression, alors *id* := *exp* est une instruction.
- exemple :
  - – a :=5*b+c est une instruction (arbre syntaxique)
- =>utilisation de grammaires

17

Analyse sémantique

- Vérifie la présence d'erreurs d'ordre sémantique (vérification de type).
- Utilisation de l'arbre résultat de l'analyse syntaxique.
- Exemple, si *a* et *b* sont des réels, *a:=b+20* est une instruction.

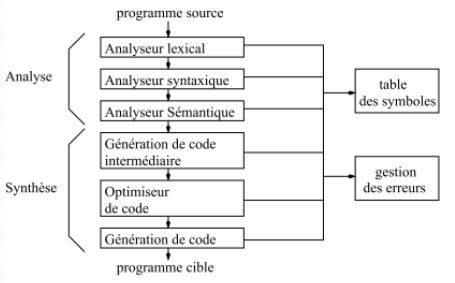


Cette modification est nécessaire car la représentation des entiers est différente de celle des réels. Autre exemple d'erreur sémantique : tableau indicé par un réel.

18

Synthèse du code

- 3 phases :
  - génération du code intermédiaire
  - optimisation du code
  - génération du code exécutable.



19

Synthèse du code

- Génération du code intermédiaire :
  - ce n'est pas obligatoire mais 2 bonnes propriétés : facile à produire et à traduire en langage machine
  - utilisation de variables temporaires
  - choix de l'ordre pour faire un calcul.
- Optimisation de code :
  - amélioration du code intermédiaire
  - réduction du nombre de variables et d'instructions.
- Production de code
  - choix des emplacements mémoire pour les variables
  - assignation de variables aux registres.

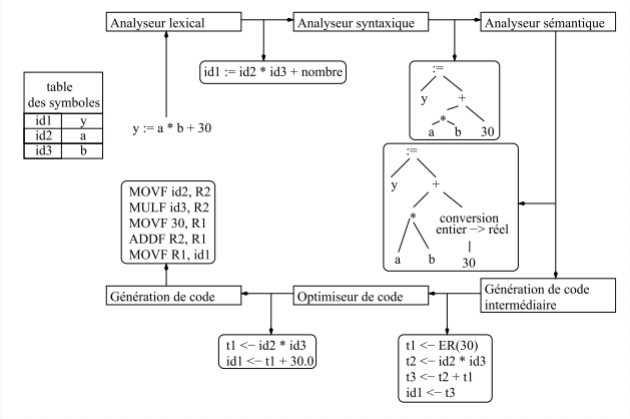
20

Table des symboles et détection des erreurs

- La table des symboles enregistre les identifiants et les attributs (emplacement mémoire, type, portée) :
  - chaque identifiant (variable) a une entrée dans la table des symboles
  - l'analyseur lexical crée dans la TS, une entrée à chaque fois qu'il rencontre un nouvel identificateur. Par contre, les attributs seront calculés plus tard.
  - L'analyseur sémantique se servira de la table des symboles pour vérifier la concordence des types.
- Détection des erreurs à plusieurs niveaux, essentiellement pendant l'analyse :
  - erreur lexicale : le flot de caractères n'est pas reconnu
  - erreur syntaxique : construction non reconnue par le langage
  - erreur sémantique : pb de typage,...

21

Exemple de traduction



22

Construction automatique d'un analyseur lexical

- L'analyseur lexical est la première composante de la phase d'analyse nécessaire à un compilateur.
- Le rôle de l'analyseur lexical:
  - lit les caractères en entrée (le prog. source) et reconnait des lexèmes
  - Le résultat est une suite de lexèmes qui propose une première structuration du programme => table des symboles
  - Il existe une interaction très forte entre l'Analyseur Lexical et l'analyseur syntaxique.

Construction automatique de l'AL à l'aide d'automates

- L'analyse lexicale permet :
  - Une simplification de la phase d'analyse.
  - Efficacité du compilateur accrue.
  - Portabilité du compilateur.
- Objectifs : construire un AL qui accepte N unités lexicales notées Lex1,…,Lexn
  - Représenter chaque unité lexicale par une expression régulière.
  - Construire un Automate Fini Non détermiste pour chaque expression régulière
  - Transformer chaque Automate Fini Non détermiste en Automate Fini détermiste (optimisation)
  - Réduire le nombre d'états des AFD (optimisation).

25

Analyse syntaxique

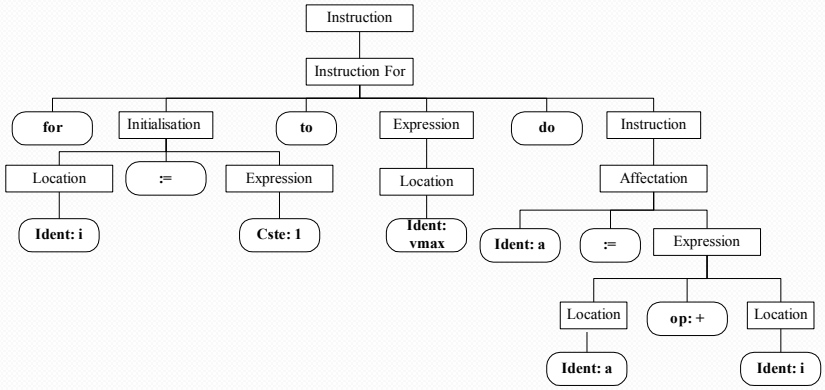
- L'analyse syntaxique est la deuxième étape de la partie analyse d'un compilateur. Elle vise à structurer les mots découverts par l'analyseur lexical sous forme de phrases. C'est l'étape la plus importante de l'analyse.

- Lors de l'analyse syntaxique, on vérifie que l'ordre des tokens correspond à l'ordre définit pour le langage. On dit que l'on vérifie la syntaxe du langage à partir de la définition de sa grammaire. Cette phase produit une représentation sous forme d'arbre de la suite des tokens obtenus lors de la phase précédente. Par exemple, l'arbre suivant représente la structure de la phrase :

26

Analyse syntaxique

- for i :=1 to vmax do a :=a+i



Analyse syntaxique

- besoin d'une grammaire
  - Le principe de base des grammaires est de donner un ensemble de règles pour engendrer les mots d'un langage. Les mots générés par une grammaire sont ceux qui peuvent être obtenus en appliquant les règles de la grammaire.

Exemple

- Soit G (alphabet, symb terminaux, règles, symb départ) la grammaire suivante :

$G = (V = \{S, a, b\}, \Sigma = \{a, b\}, R = \{S \rightarrow \epsilon, S \rightarrow aSb\}, S)$.

G définit le langage $\{a^n b^b, n \geq 0\}$ avec un seul non terminal S qui est aussi le symbole de départ.

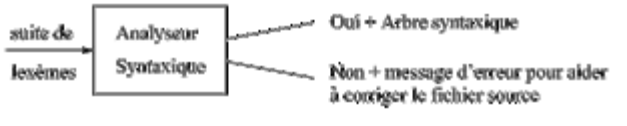
Le mot $aabb$ fait partie du langage défini par cette grammaire car :

S	
aSb	$(S \rightarrow aSb)$
$aaSbb$	$(S \rightarrow aSb)$
$aabb$	$(S \rightarrow \epsilon)$

29

Analyse syntaxique

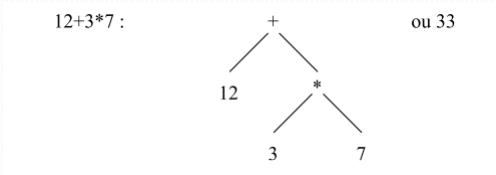
- Il est possible de construire automatiquement un analyseur syntaxique à partir d'une grammaire.



30

Analyse sémantique

- L'analyse syntaxique n'est pas suffisante :
- Dans cette phase on vérifie que les variables ont un type correct. Par exemple, il faut vérifier que la variable 'i' possède bien le type 'entier', et que la variable 'a' est bien un nombre. Cette opération s'effectue en parcourant l'arbre syntaxique et en vérifiant à chaque niveau que les opérations sont correctes.



31

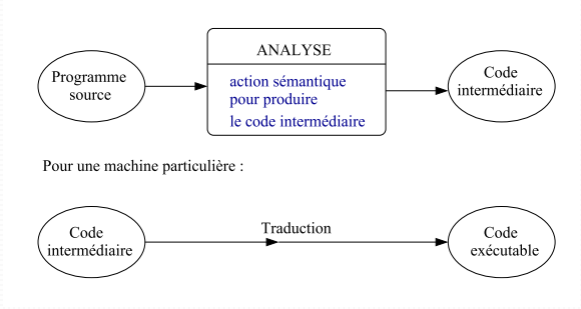
Analyse sémantique

- Rôle de l'analyse sémantique:
  - Entrée :arbre de syntaxe abstraite issu de l'analyse syntaxique
  - Sortie : arbre de syntaxe abstraite décoré: résultat de la compréhension du programme: une erreur si le programme est incorrect
- Repérer les utilisations d'objets non déclarés. Relier les déclarations d'objets(variables, types, procédures, fonctions …) et leur utilisation.
- Une fois ce lien réalisé, le sauvegarder pour les autres phases.

32

Génération du code intermédiaire

- On peut comparer le code intermédiaire à un assembleur universel (indépendant de la machine cible) :



33

Génération du code intermédiaire

- Les arbres abstraits sont déjà des représentations intermédiaires.
- =>passer d'une représentation arborescente à une représentation linéaire.

- Un quadruplet est une structure composée de 4 champs appelés op, arg1, arg2 et résultat.
  - x := y + z , on associe le quadruplet : +, y, z, x

34

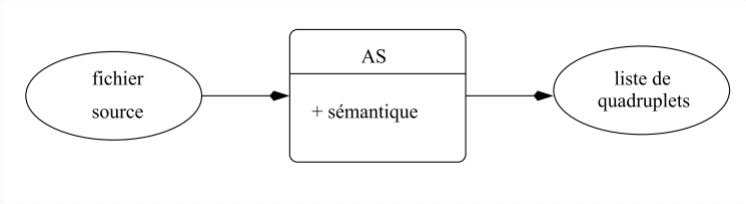
Génération du code intermédiaire

for i :=1 to vmax do a :=a+i

```
var_a                                   A0000                   ; les étiquettes des variables
        var_i                                           A0001
        var_vmax                        A0002

; le code du programme
        mov var_i,1
loop:
        mov Ao, (var_i)
; comparaison i >= vmax
        jge Ao, (var_vmax), finFor      ; si vrai aller en finFor
        mov Ao, (var_a)
; calcul de a+i
        add Ao, Ao, (var_i)
        mov var_a,Ao                                            ; a := a+i
        mov Ao, (var_i)
; on incrémente i
        add Ao, Ao, 1
        mov var_i, 1
        jmp loop                                                        ; et on continue la boucle
```


Génération de quadruplets dirigée par la syntaxe



- Principe : pour chaque nœud interne de l'arbre, on a besoin de créer des variables temporaires (car prochains quadruplets ne sont pas connus)

Lex et Yacc

Lex et Yacc sont des outils très populaires de génération d'analyseurs lexicaux (Lex) et syntaxiques (Yacc).

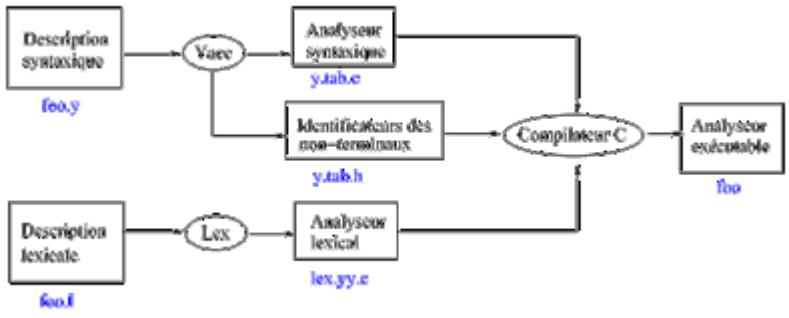
- LEX générateur d'analyseur lexical.
  - Prend en entrée la définition des unités lexicales.
  - Produit un automate fini déterministe minimal permettant de reconnaître les unités lexicales.
  - L'automate est produit sous la forme d'un programme C.
- YACC générateur d'analyseur syntaxique.
  - Prend en entrée la définition d'un schéma de traduction (grammaire + actions sémantiques).
  - Produit un analyseur syntaxique pour le schéma de traduction.
  - L'analyseur est produit sous la forme d'un programme C.

37

Lex et Yacc

38

Lex et Yacc



39