MINISTERE DE L'ENSEIGNEMENT SUPERIEUR
ET DE LA RECHERCHE SCIENTIFIQUE
BADJI MOKHTAR-ANNABA UNIVERSITY
UNIVERSITE BADJI MOKHTAR-ANNABA

وزارة التعليم العالي و البحث العلمي
جامعة باجي مختار – عنابـــــــــة

FACULTE DES SCIENCES DE L'INGENIORAT
DEPARTEMENT D'INFORMATIQUE

كلية علـــــــــوم الهندســـــــة
قسم الإعـــــــلام الألـــــــــي

COMPILATION

3ème Année Licence en Informatique

SUPPORT DE COURS REALISE PAR
PR SOUICI-MESLATI LABIBA
souici_labiba@yahoo.fr

2012-2013

SYLLABUS (DESCRIPTIF OFFICIEL)

Domaine: Mathématique et Informatique
Filière: Informatique
Spécialité: Licence Informatique (L3)
Semestre: 1, Année 2012-2013
Unité d'enseignement: UEI13
Matière: COMPILATION
Nombre de Crédits: 6
Volume horaire hebdomadaire total : 6h00

Cours : 3h **TD** : 1h30 **TP** : 1h30

Evaluation **Examen final** : 50%
TD : 25% (50% : présence et participation et 50% : micro interrogation)
TP : 25% (50% : présence et participation et 50% : évaluations sur machine)

Objectifs

1. Compréhension du cheminement d'un programme (texte) source vers un programme (code).
2. Etude des étapes du processus de compilation d'un langage évolué.
3. Etude de méthodes et techniques utilisées en analyse lexicale, syntaxique et sémantique.
4. Familiarisation, en TP, avec des outils de génération d'analyseurs lexicaux et syntaxiques (LEX et YACC).

Contenu

1 Introduction à la compilation
- Les différentes étapes de la compilation
- Compilation, interprétation, traduction

2 Analyse Lexicale
- Expressions régulières
- Grammaires
- Automates d'états finis
- Un exemple de générateur d'analyseurs lexicaux : LEX

3 Analyse Syntaxique
- Définitions : grammaire syntaxique, récursivité gauche, factorisation d'une grammaire, grammaire ε-libre
- Calcul des ensembles des débuts et suivants
- Méthodes d'analyse descendante : la descente récursive, LL(1)
- Méthodes d'analyse ascendante : SLR(1), LR(1), LALR(1) (méthode des items)
- Un exemple de générateur d'analyseurs syntaxiques : YACC

4 Traduction dirigée par la syntaxe (Analyse sémantique)

5 Formes intermédiaires
- Forme postfixée
- Quadruplés
- Triplés directs et indirects
- Arbre abstrait

6 Allocation - Substitution – Organisation des données à l'exécution

Références Bibliographiques

Ouvrages existants au niveau de la bibliothèque de l'université, la référence 1 est vivement recommandée

1. Aho A., Sethi R., Ullman J., "Compilateurs : Principes, techniques et outils", Inter-éditions, 1991 et Dunod, 2000
2. Drias H., "Compilation: Cours et exercices", OPU, 1993
3. Wilhem R., Maurer D., "Les compilateurs: Théorie, construction, génération", Masson, 1994

CHAPITRE 1 : INTRODUCTION AUX COMPILATEURS

1.1 Définition

Un compilateur est un programme qui a comme entrée un code source écrit en langage de haut niveau (langage évolué) est produit comme sortie un code cible en langage de bas niveau (langage d'assemblage ou langage machine).

La traduction ne peut être effectué que si le code source est correct car, s'il y a des erreurs, le rôle du compilateur se limitera à produire en sortie des messages d'erreurs (voir figure 1.1).

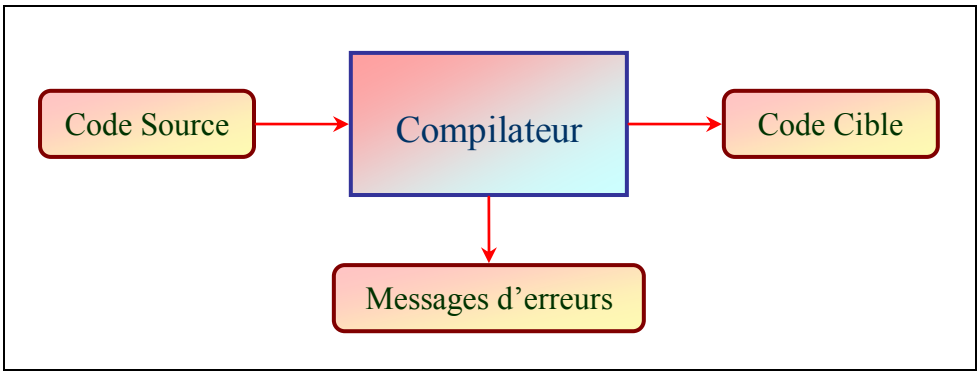

Figure 1.1. Rôle du compilateur

Un compilateur est donc un traducteur de langage évolué qu'on ne doit pas confondre avec un interpréteur qui est un autre type de traducteur. En effet, au lieu de produire un programme cible comme dans le cas d'un compilateur, un interpréteur exécute lui-même au fur et à mesure les opérations spécifiées par le programme source. Il analyse une instruction après l'autre puis l'exécute immédiatement. A l'inverse d'un compilateur, il travaille simultanément sur le programme et sur les données. L'interpréteur doit être présent sur le système à chaque fois que le programme est exécuté, ce qui n'est pas le cas avec un compilateur. Généralement, les interpréteurs sont assez petits, mais le programme est plus lent qu'avec un langage compilé. Un autre inconvénient des interpréteurs est qu'on ne peut pas cacher le code, et donc garder des secrets de fabrication : toute personne ayant accès au programme peut le consulter et le modifier comme elle le veut. Par contre, les langages interprétés sont souvent plus simples à utiliser et tolèrent plus d'erreurs de codage que les langages compilés. Des exemples de langages interprétés sont : BASIC, scheme, CaML, Tcl, LISP, Perl, Prolog

Il existe des langages qui sont à mi-chemin de l'interprétation et de la compilation. On les appelle langages P-code ou langages intermédiaires. Le code source est traduit (compilé) dans une forme binaire compacte (du pseudo-code ou p-code) qui n'est pas encore du code machine. Lorsqu'on exécute le programme, ce P-code est interprété. Par exemple en Java, le programme source est compilé pour obtenir un fichier (.class) « byte code » qui sera interprété par une machine virtuelle. Un autre langage p-code : Python.

Les interpréteurs de p-code peuvent être relativement petits et rapides, si bien que le p-code peut s'exécuter presque aussi rapidement que du binaire compilé. En outre les langages p-code peuvent garder la flexibilité et la puissance des langages interprétés.

1.2 Structure générale d'un compilateur

Un compilateur est généralement composé de modules correspondant aux phases logiques de l'opération de compilation (voir figure 1.2).

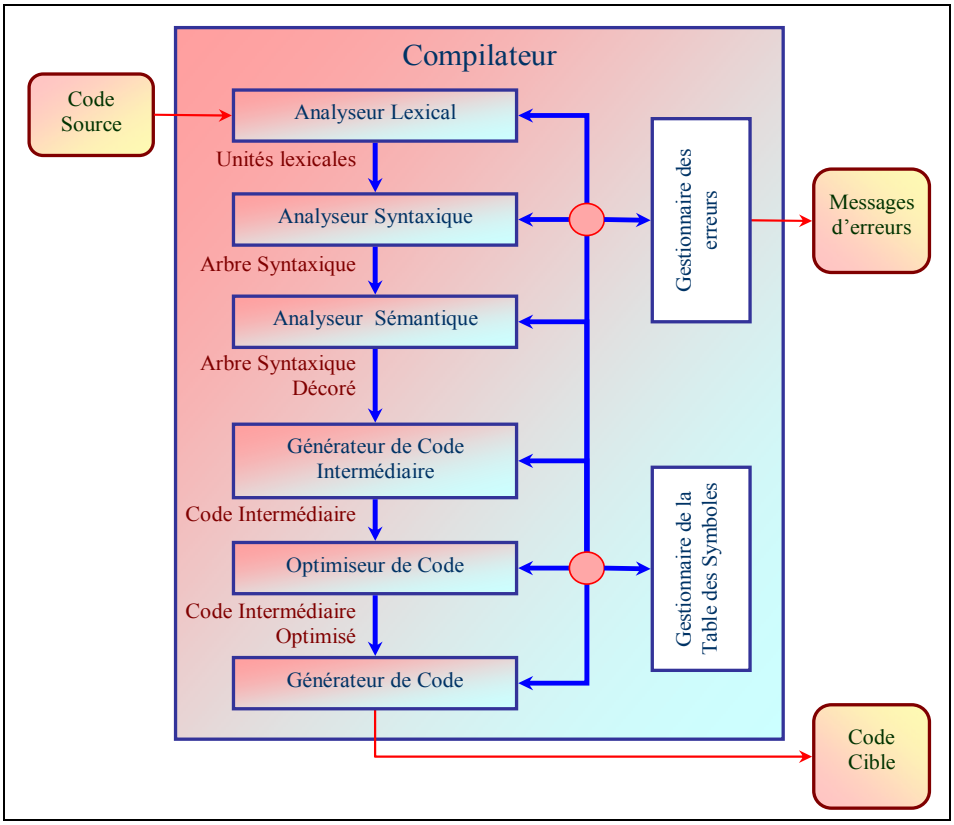

Figure 1.2. Phases et modules de compilation

Chacun des modules de la figure 1.2 (excepté les modules gestionnaires de table de symboles et d'erreurs), représente une phase logique qui reçoit en entrée une représentation de code source et la transforme en une autre forme de représentation.

La structure représentée par la figure 1.2 est purement conceptuelle. Elle correspond à l'organisation logique d'un compilateur. En pratique, plusieurs phases peuvent être regroupées en une seule passe qui reçoit en entrée une certaine représentation et donne en sortie une autre.

Par exemple, les phases d'analyses lexicale, syntaxique, sémantique et la génération du code intermédiaire peuvent correspondre à seule passe dans laquelle l'analyseur syntaxique est le module maître qui appelle, à chaque fois, l'analyseur lexical pour obtenir une unité lexicale, puis déterminer graduellement la structure syntaxique du code et enfin appelle le générateur de code qui effectue l'analyse sémantique et produit une partie du code intermédiaire.

1.2.1 L'analyseur lexical

Connu aussi sous l'appellation **Scanner**, l'analyseur lexical a pour rôle principal la lecture du texte du code source (suite de caractères) puis la formation des unités lexicales (appelées aussi entités lexicales, lexèmes, jetons, tokens ou encore atomes lexicaux).

Exemple

Considérons l'expression d'affectation **a := b + 2 * c** ;

Les unités lexicales qui apparaissent dans cette expression sont :

Unité lexicale	Sa nature
a	Identificateur de variable
:=	Symbole d'affectation
b	Identificateur de variable
+	Opérateur d'addition
2	Valeur entière
*	Opérateur de multiplication
c	Identificateur de variable
;	Séparateur

L'analyseur a aussi comme rôle l'élimination des informations inutiles pour l'obtention du code cible, le stockage des identificateurs dans la table des symboles et la transmission d'une entrée à l'analyseur syntaxique. Concernant les informations inutiles, il s'agit généralement du caractère espace et des commentaires.

1.2.2 L'analyseur syntaxique

L'analyseur syntaxique (appelé Parser en anglais) a pour rôle principal la vérification de la syntaxe du code en regroupant les unités lexicales suivant des structures grammaticales qui permettent de construire une représentation syntaxique du code source. Cette dernière a souvent une structure en arbre. Notons que durant cette phase, des informations, telles que le type des identificateurs, sont enregistrées dans la table des symboles

Exemple

La représentation sous forme d'arbre syntaxique de l'expression « **a := b + 2 * c** ; » est donnée par la figure 1.3. Dans cette structure d'arbre, les nœuds représentent des opérateurs et les feuilles de l'arbre représentent les valeurs et les variables sur lesquelles s'effectuent les opérations. La figure donne aussi le parcours qui est fait sur cet arbre lors de l'évaluation. D'autre parcours peuvent être envisagés pour réaliser différentes tâches, cependant ces parcours ont lieu dans les phases ultérieures de la compilation.

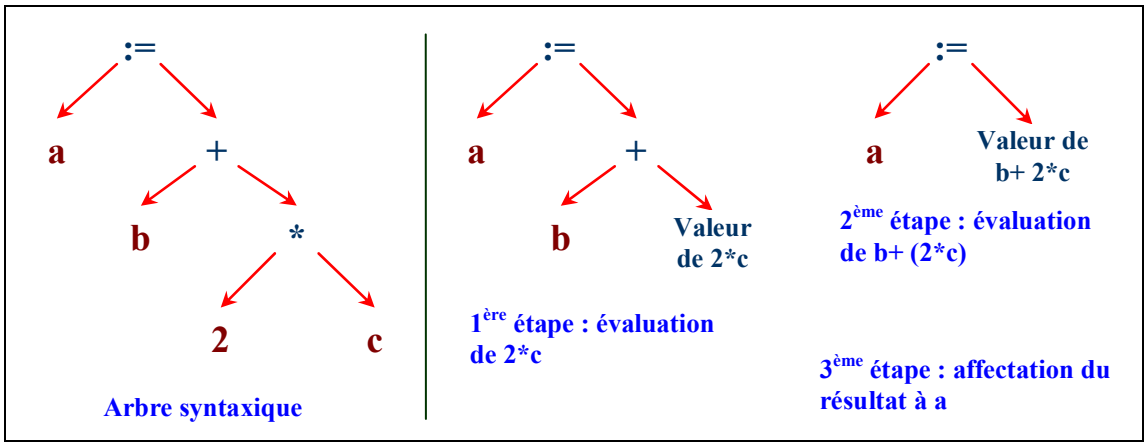

Figure 1.3. Arbre syntaxique et parcours d'évaluation

1.2.3 L'analyseur sémantique

Il comme rôle principal le contrôle du code source, pour détecter éventuellement l'existence d'erreurs sémantiques, et la collecte des informations destinées à la production du code intermédiaire. Un des constituants importants de la phase d'analyse sémantique est le contrôle du type qui consiste à vérifier si les opérandes de chaque opérateur sont conformes aux spécifications du langage utilisé pour le code source.

Exemple

Dans l'analyse sémantique de « **a := b + 2 * c** ; », il faut vérifier que, si a est de type entier, alors b et c le sont aussi, sinon il faut signaler une erreur.

Si on suppose que a, b et c sont de type réel, alors pendant l'évaluation de l'expression, l'analyse sémantique aura comme tâche d'insérer une opération de conversion de type pour transformer la valeur entière 2 en valeur réelle 2.0. Cela peut être effectué sur l'arbre syntaxique comme le montre la figure 1.4 (arbre syntaxique décoré).

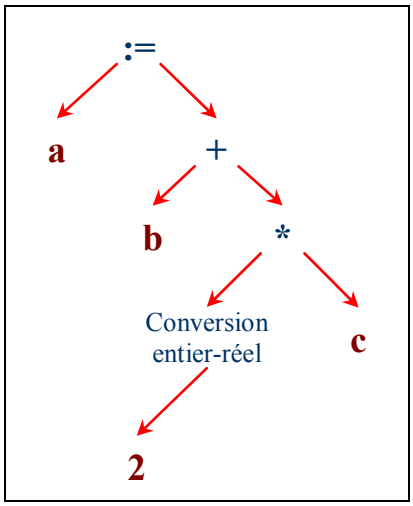

Figure 1.4. Enrichissement de l'arbre syntaxique lors de la phase d'analyse sémantique

1.2.4 Le générateur de code intermédiaire

Certains compilateurs construisent explicitement une représentation intermédiaire du code source sous forme d'un code intermédiaire qui n'est pas directement exécutable par une machine spécifique. C'est plutôt un code généré pour une machine abstraite (virtuelle), qui a la double caractéristique d'être, à la fois, facile à produire, à partir de l'arbre syntaxique, et facile à convertir pour une machine réelle donnée.

Exemple

Nous avons supposé que la machine abstraite est une machine à une adresse qui dispose d'un seul accumulateur et dont le jeu d'instruction contient les instructions LoadValue, ConvReal, Mul, Add et Store. Elles permettent de réaliser les opérations données dans la deuxième colonne. Nous avons supposé aussi que **a** occupe la première entrée dans la table des symboles, **b** occupe la deuxième et **c** la troisième.

Ainsi, le code intermédiaire de l'expression « **a := b + 2 * c** ; », peut être comme suit :

Code	**Signification opérationnelle des instructions**
LoadValue 2	Charger l'accumulateur avec une valeur directe (2)
ConvReal	Convertir le contenu de l'accumulateur en réel
Store temp1	Stocker le résultat dans la variable temporaire temp1
Load temp1	Charger l'accumulateur avec la valeur de temp1
Mul 3	Multiplier le contenu de l'accumulateur par le contenu de l'entrée 3 de la table de symboles (c)
Add 2	Additionner le contenu de l'accumulateur au contenu l'entrée 2 de la table de symboles (b)
Store 1	Ranger le contenu de l'accumulateur dans l'entrée 1 de la table de symboles (a)

1.2.5 L'optimiseur du code intermédiaire

Lors de la phase d'optimisation, le code intermédiaire est changé pour améliorer les performances du code cible qui en sera généré. Il s'agit principalement de réduire le temps d'exécution et l'espace mémoire qui sera occupé par le code cible. L'optimisation supprime, par exemple, les identificateurs non utilisés, élimine les instructions inaccessibles, élimine les instructions non nécessaires, fait ressortir hors des boucles les instructions qui ne dépendent pas de l'indice de parcours des boucles, etc.

L'optimisation risque de ralentir le processus de compilation dans son ensemble mais elle peut avoir un effet positif considérable sur le code cible qui sera généré ultérieurement.

Exemple

On constate dans l'exemple précédent que la valeur convertie 2.0 est stockée dans temp1 puis récupérée et chargée dans l'accumulateur. Puisque aucun usage n'est fait de temp1 dans le reste du code, il est possible d'éliminer les deux instructions en question. Après conversion, le résultat 2.0 reste alors dans l'accumulateur et sera utilisé directement dans la multiplication. Noter que, à l'opposé, si la conversion en réel de la valeur 2 est souvent nécessaire, il serait préférable de la stocker dans un espace temporaire. Cependant, ce dernier augmente l'espace réservé aux données dans le code cible.

Le nouveau code intermédiaire est le suivant :

Code	Signification opérationnelle des instructions
LoadValue 2	Charger l'accumulateur avec une valeur directe
ConvReal	Convertir le contenu de l'accumulateur en réel
Mul 3	Multiplier le contenu de l'accumulateur par le contenu de l'entrée 3 de la table de symboles
Add 2	Additionner le contenu de l'accumulateur au contenu l'entrée 2 de la table de symboles
Store 1	Ranger le contenu de l'accumulateur dans l'entrée 1 de la table de symboles

1.2.6 Le générateur du code cible

C'est la phase finale d'un compilateur qui consiste à produire du code cible dans un langage d'assemblage ou un langage machine donné. Le code généré est directement exécuté par la machine en question ou alors il l'est après une phase d'assemblage.

Exemple

Considérons une machine à deux adresses qui dispose de deux registres de calcul R1 et R2. Nous supposons que les variables a, b et c de la table des symboles ont comme adresses de cellules mémoires correspondantes ad1, ad2 et ad3. Le code cible qui sera généré pour cette machine est donné dans le tableau suivant :

Code	Signification opérationnelle des instructions
MOV R1, #2	Charger R1 avec la valeur directe 2
CReal R1	Convertir le contenu de R1 en réel
MOV R2, ad3	Charger le registre R2 avec le contenu de la cellule mémoire d'adresse ad3
MUL R1, R2	Multiplier le contenu de R1 par le contenu de R2, le résultat est dans le premier registre
MOV R2, ad2	Charger R2 avec le contenu de l'adresse ad2
ADD R1, R2	Additionner le contenu de R1 au contenu de R2, le résultat est dans le premier registre
STO R1, ad1	Ranger le contenu de R1 dans la cellule mémoire d'adresse ad1

1.2.7 Le gestionnaire de la table de symbole

Les phases logiques de compilation échangent des informations par l'intermédiaire de la table des symboles. C'est une structure de données (généralement une table) contenant un enregistrement pour chaque identificateur utilisé dans le code source en cours d'analyse. L'enregistrement contient, parmi d'autres informations, le nom de l'identificateur, son type, et l'emplacement mémoire qui lui correspondra lors de l'exécution.

A chaque fois que l'analyseur lexical rencontre un identificateur pour la première fois, le gestionnaire de la table des symboles insère un enregistrement dans la table et l'initialise avec les informations actuellement disponibles (le nom). Lors de l'analyse syntaxique, le gestionnaire associera le type à l'identificateur, alors que, lors de l'analyse sémantique, une vérification de types est opérée grâce à cet enregistrement.

1.2.8 Le gestionnaire des erreurs

Son rôle est de signaler les erreurs qui peuvent exister dans le code source et qui sont détectées lors des différentes phases logiques de la compilation. Il doit produire, pour chaque erreur, un diagnostic clair et sans ambiguïté qui permettra la localisation et la correction de l'erreur par l'auteur du code source.

1.3 Outils de construction des compilateurs

Suite au développement des premiers compilateurs, on s'est très vite rendu compte que certaines tâches liées au processus de compilation peuvent être automatisées, ce qui facilite grandement la construction des compilateurs. La notion d'outils de construction de compilateurs est alors apparue.

La première catégorie est représentée par des outils généraux appelés Compilateurs de Compilateurs, Générateurs de Compilateurs ou Systèmes d'écriture de traducteurs. Les outils de cette catégorie se sont souvent orientés vers un modèle particulier de langages et ne sont adaptés qu'à la construction de compilateurs pour des langages correspondant à ce modèle.

La deuxième catégorie correspond à des outils spécialisés dans la construction automatique de certaines phases d'un compilateur, tels que :

- Les constructeurs ou générateurs automatiques d'analyseurs lexicaux à partir d'une spécification contenant des expressions régulières
- Constructeurs ou générateurs automatiques d'analyseurs syntaxiques à partir d'une spécification basée sur une grammaire non contextuelle et en utilisant des algorithmes d'analyse puissants mais difficile à mettre en œuvre manuellement.