Chapitre 0 : Rappels de Compilation (Semaine 1)

Etienne Kouokam

Département d'Informatique
Université de Yaoundé I, Cameroun

Année académique 2019-2020
Univ de Ydé I : Mars-Juin 2020

Plan

Objectifs

1. Ce cours est une suite au cours Théorie des Langages & Compilation, fait au niveau 3.
2. Il présente quelques méthodes mathématiques de l'informatique théorique qui pourront servir dans la compilation.
3. L'objectif, ici, est de comprendre ce qu'est un automate à pile et quel son lien avec les grammaires algébriques.
4. Cette étape supplémentaire permettra aussi d'aborder les méthodes d'analyses (ascendantes et descendantes) puis ultérieurement l'étude des machines de Turing.

Démarche

Montrer la démarche scientifique et de l'ingénieur qui consiste à

1. Comprendre les outils (mathématiques / informatiques) disponibles pour résoudre le problème
2. Apprendre à manipuler ces outils
3. Concevoir à partir de ces outils un système (informatique)
4. Implémenter ce système

Les outils ici sont :

- les formalismes pour définir un langage ou modéliser un système
- les générateurs de parties de compilateurs

Prérequis + Bibliographie

Prérequis : Maîtrise du cours de théorie des langages et Compilation

Bibliographie : On pourra consulter les ouvrages suivants :

1. J. E. Hopcroft, R. Motwani, and J. D. Ullman. Introduction to Automata Theory, Languages and Computation. Addison Wesley, 2001.
2. A. Aho and J. D. Ullman. Concepts fondamentaux de l'Informatique. Dunod, 1993.
3. A. Aho, R. Sethi, and J. D. Ullman. Compilateurs Principes, techniques et outils. InterEditions, 1991.
4. Terence Parr. The Definitive ANTLR Reference. The Pragmatic Programmers, 2007.

Définition (Compilation)

Traduction $C = f(L_C, L_S \rightarrow L_O)$ d'un langage dans un autre où

- L_C le langage avec lequel est écrit le compilateur
- L_S le langage source à compiler
- L_O le langage cible ou langage objet : celui vers lequel il faut traduire

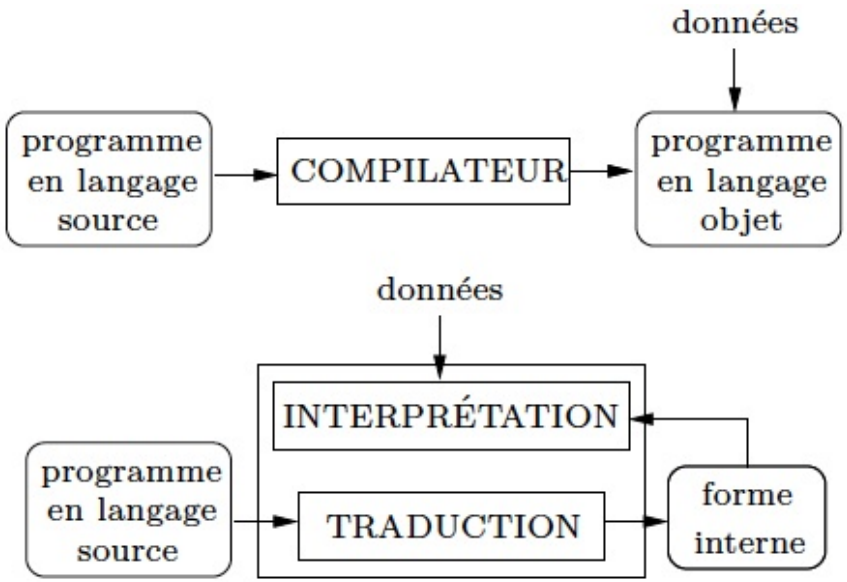


FIGURE – Compilateur Vs Interprète

Compilateur modulaire

- partie avant (analyse) : analyses lexicale, syntaxique, sémantique
- partie arrière (synthèse) : optimisation, production de code
- avantages de cette décomposition : m parties avant + n parties arrières permettent d'avoir $m \times n$ compilateurs
- le langage objet peut être celui d'une machine virtuelle (JVM . . .) : le programme résultant sera portable
- on peut interpréter la représentation intermédiaire

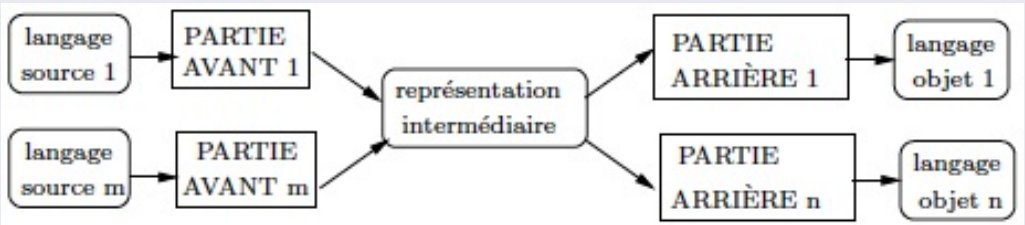


FIGURE – Compilateur modulaire

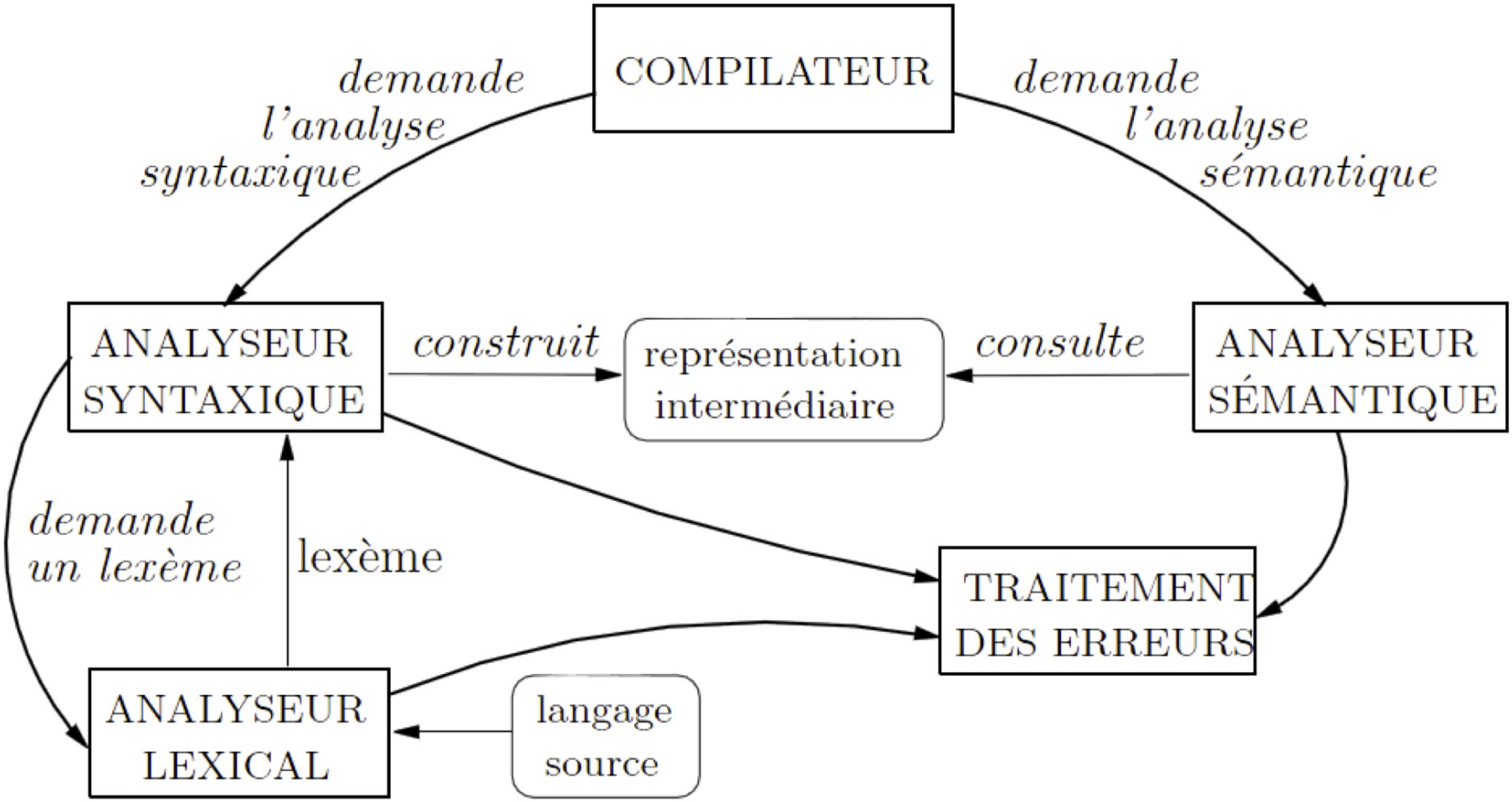


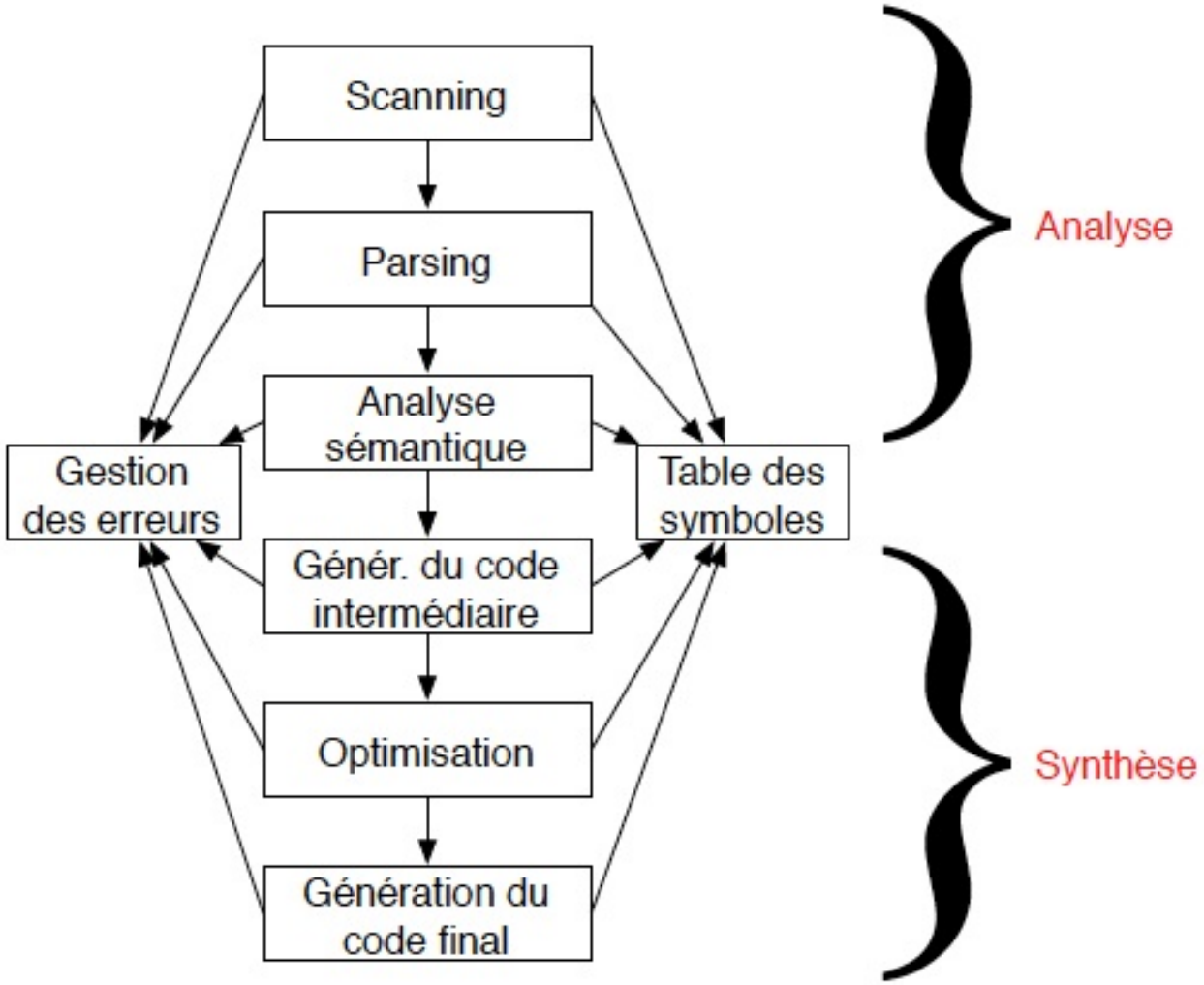
FIGURE – Schéma synthétique de la partie avant

Concepts et structures de données utilisés

- analyse lexicale : langages réguliers, expressions régulières, automates finis pour l'essentiel, mais aussi tables d'adressage dispersé, arithmétique
- **analyse syntaxique : grammaires hors-contexte, automates à pile (analyseurs descendants ou ascendants), attributs sémantiques**
- analyse sémantique : diverses sémantiques formelles (mais l'analyse sémantique est souvent codée à la main), équations de type, table de symboles
- représentation intermédiaire : arbre ou graphe le plus généralement

Les Phases de la Compilation

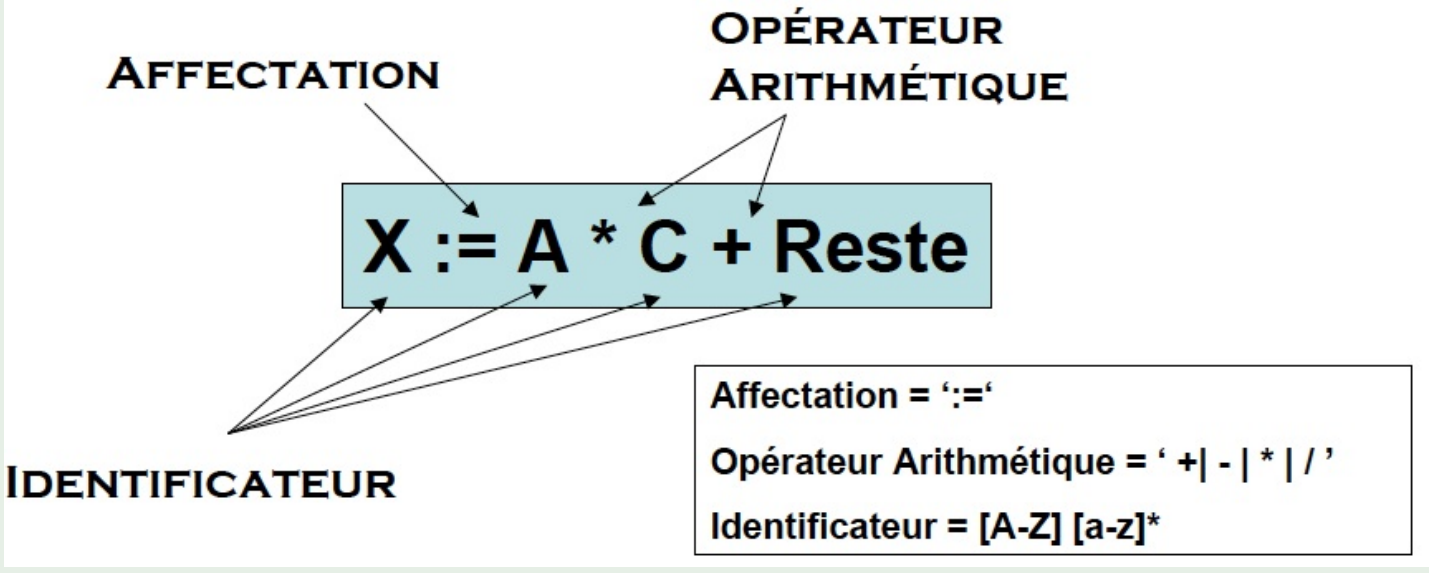
Illustration



Compilation découpée en 2 étapes

1. L'analyse décompose et identifie les éléments et relations du programme source et construit son image (représentation hierachique du programme avec ses relations),
2. La synthèse qui construit à partir de l'image un programme en langage cible

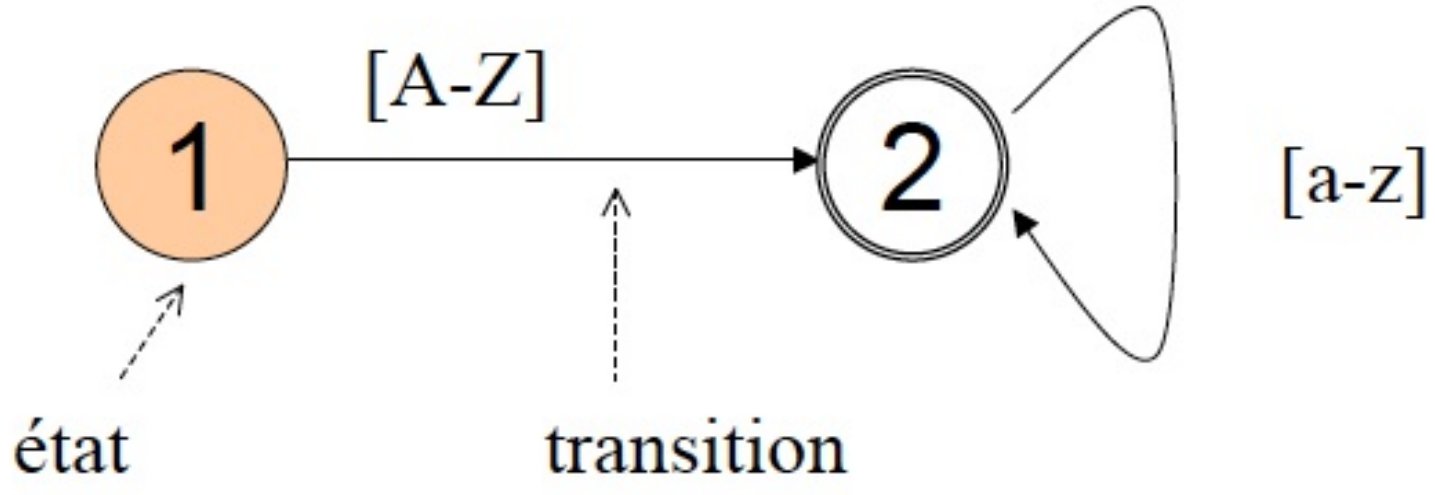
Exemple



Analyse lexicale (Scanning)

Les unités lexicales sont reconnues à l'aide d'automates

Identificateur = [A-Z] [a-z]*



Analyse syntaxique (Parsing)

- Le rôle principal de l'analyse syntaxique est de trouver la structure de la "phrase" ? (le programme) : i-e de construire une représentation interne au compilateur et facilement manipulable de la structure syntaxique du programme source.
- Le parser construit l'arbre syntaxique correspondant au code.

L'ensemble des arbres syntaxiques possibles pour un programme est défini grâce à une grammaire (context-free).

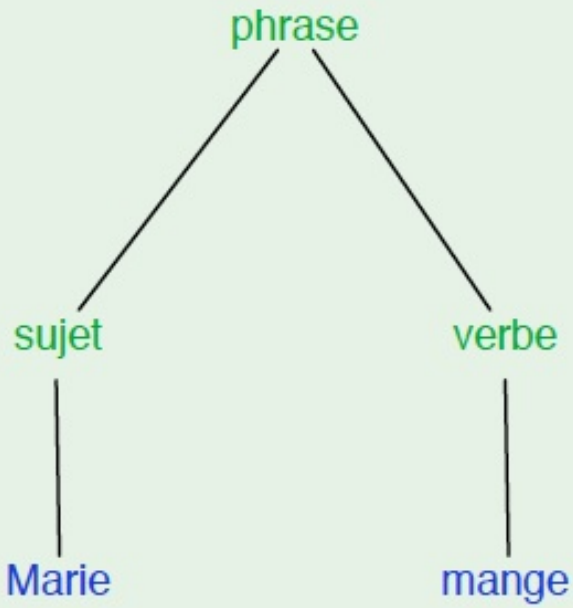
Exemple de grammaire

Exemple (grammaire d'une phrase)

- phrase = sujet verbe
- sujet = "**Jean**" | "**Marie**"
- verbe = "**mange**" | "**parle**"

peut donner

- **Jean mange**
- **Jean parle**
- **Marie mange**
- **Marie parle**



Arbre syntaxique de la phrase
Marie mange

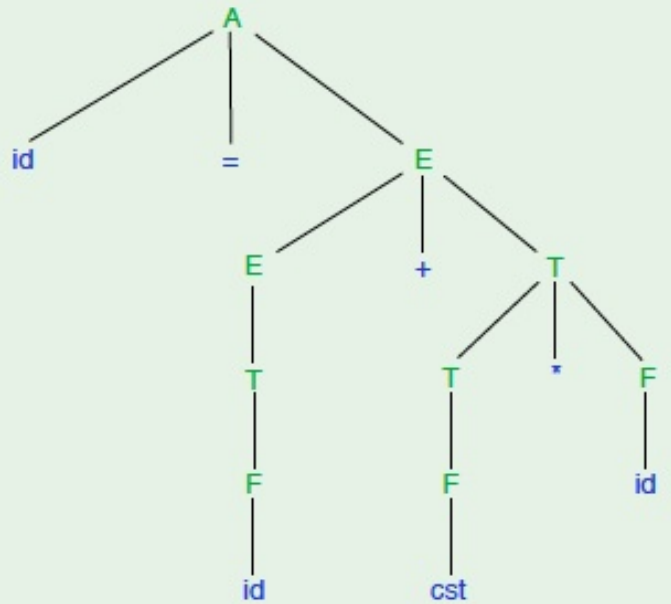
Exemple de grammaire

Exemple (grammaire d'une expression)

- A = "**id**" "**=**" E
- E = T | E "**+**" T
- T = F | T "*****" F
- F = "**id**" | "**cst**" | "**(**" E "**)**"

peut donner :

- **id = id**
- **id = id + cst * id**
- ...



Arbre syntaxique de la phrase
id = id + cst * id

Exemple de grammaire (suite)

Exemple (grammaire d'une expression)

- A = "**id**" "**=**" E
- E = T | E "**+**" T
- T = F | T "*****" F
- F = "**id**" | "**cst**" | "**(**" E "**)**"

peut donner :

- **id = id**
- **id = id + cst * id**
- . . .

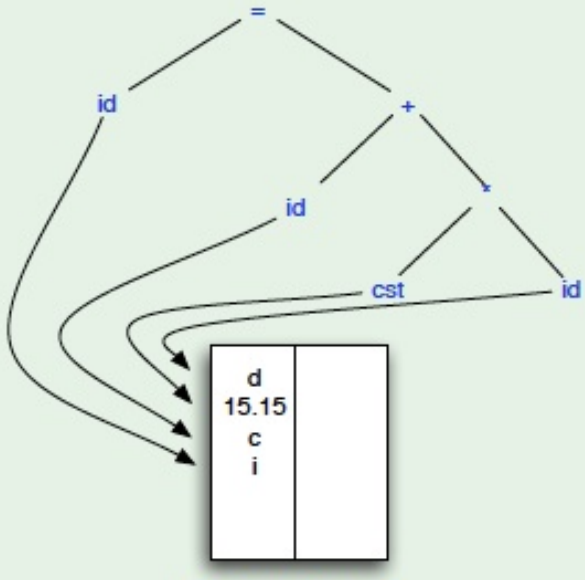


Table des symboles

Arbre syntaxique abstrait avec références à la table des symboles de la phrase **i = c + 15.15 * d**

Analyse sémantique

Rôle de l'analyse sémantique

Pour un langage impératif, l'analyse sémantique (appelée aussi gestion de contexte) s'occupe des relations non locales ; elle s'occupe ainsi :

1. du contrôle de visibilité et du lien entre les définitions et utilisations des identificateurs (en utilisant/construisant la table des symboles)
2. du contrôle de type des objets, nombre et type des paramètres de fonctions
3. du contrôle de flot (vérifie par exemple qu"un goto est licite - voir exemple plus bas)
4. de construire un arbre syntaxique abstrait complété avec des informations de type et un graphe de contrôle de flot pour préparer les phases de synthèse.

Exemple de grammaire (suite)

Exemple (pour l'expression $i = c + 15.15 * d$)

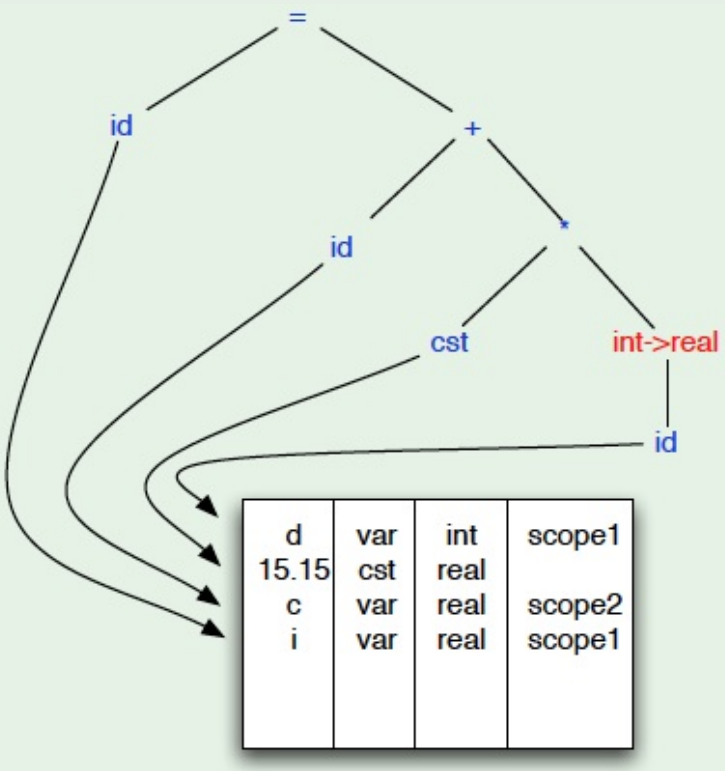


Table des symboles

Arbre syntaxique abstrait modifié avec références à la table des symboles de la phrase **i = c + 15.15 * d**

Synthèse

Phase de synthèse

Pour un langage impératif, la synthèse comporte les 3 phases

1. Génération de code intermédiaire sous forme de langage universel qui
   - utilise un adressage symbolique
   - utilise des opérations standard
   - effectue l'allocation mémoire (résultat dans des variables temporaires...)
2. Optimisation du code
   - supprime le code "mort"
   - met certaines instructions hors des boucles
   - supprime certaines instructions et optimise les accès à la mémoire
3. Production du code final
   - Allocation de mémoire physique
   - gestion des registres

Exemple de grammaire (suite) (pour le code $i = c + 15.15 * d$)

1. Génération de code intermédiaire

$$
\begin{array}{lcr}
\text{temp1} & \leftarrow & 15.5 \\
\text{temp2} & \leftarrow & \mathit{Int2Real}(\mathit{id}3) \\
\text{temp2} & \leftarrow & \mathit{temp}1 * \mathit{temp}2 \\
\text{temp3} & \leftarrow & \mathit{id}2 \\
\text{temp3} & \leftarrow & \mathit{temp}3 + \mathit{temp}2 \\
\text{id1} & \leftarrow & \mathit{temp}3
\end{array}
$$

2. Optimisation du code

$$
\begin{array}{lcr}
\text{temp1} & \leftarrow & \mathit{Int2Real}(\mathit{id}3) \\
\text{temp1} & \leftarrow & 15.15 * \mathit{temp}1 \\
\text{id1} & \leftarrow & \mathit{id}2 + \mathit{temp}1
\end{array}
$$

3. Production du code final

Exemple de grammaire (suite) (pour le code $i = c + 15.15 * d$)

1. Génération de code intermédiaire
2. Optimisation du code

$$\begin{array}{lcl} \text{temp1} & \leftarrow & \mathit{Int2Real}(\mathit{id3}) \\ \text{temp1} & \leftarrow & 15.15 * \mathit{temp1} \\ \text{id1} & \leftarrow & \mathit{id2} + \mathit{temp1} \end{array}$$

3. Production du code final

MOVF	id3, R1
ITOR	R1
MULF	15.15, R1, R1
ADDF	id2, R1, R1
STO	R1, id1