Introduction à la Compilation

Frédéric Béchet
Carlos Ramisch
Sylvain Sené [1]

Compilation – L3 Informatique
Département Informatique et Interactions
Aix Marseille Université

1. Adapté des diapos de Alexis Nasr

Infos pratiques

- 10 cours, 10 séances de TD et 10 séances de TP
- Emploi du temps :
  - Consulter l'ENT
- Évaluation :
  - Partiel (33.33%)
  - Projet (33.33%) – 2 parties
  - Examen final (33.33%)
- Page web du cours : Ametice

Bibliographie

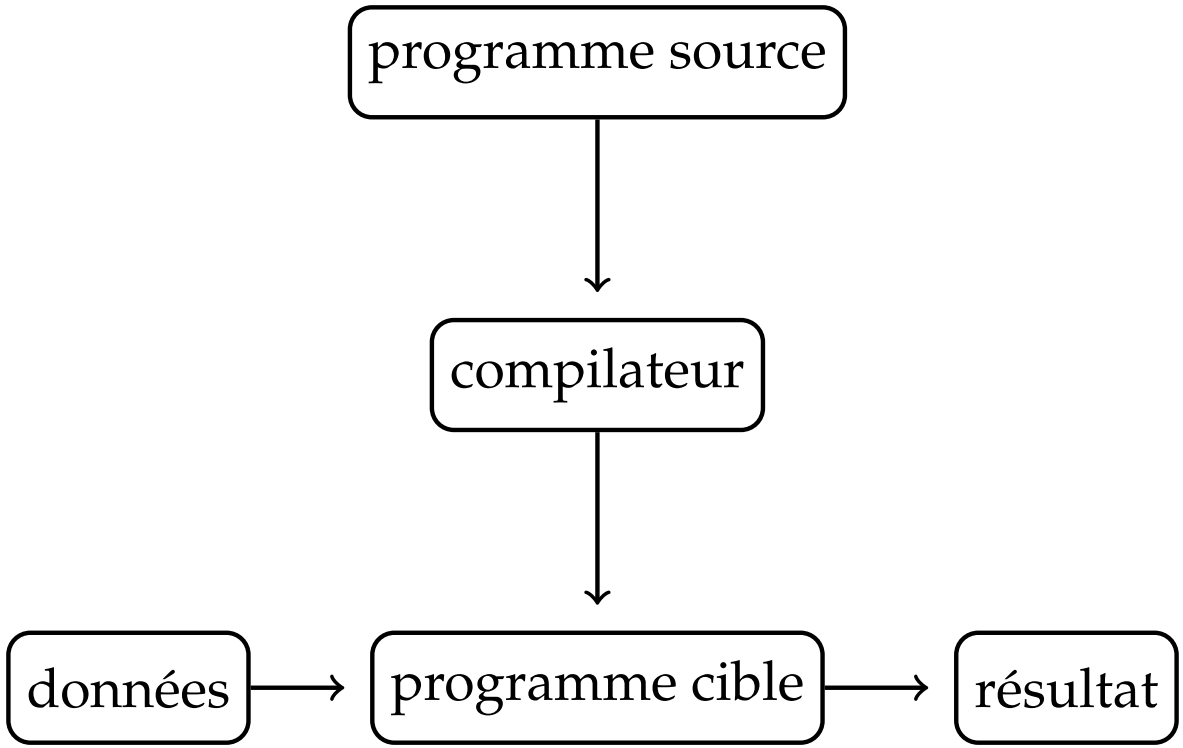
- Alfred Aho, Monica Lam, Ravi Sethi et Jeffrey Ullman **Compilateurs principes, techniques et outils**, 2ème édition. Pearson Education, 2007
- Andrew Appel **Modern compiler implementation in C**. Cambridge University Press, 1998
- John Hopcroft, Rajeev Motwani, Jeffrey Ullman **Introduction to Automata Theory, Languages and Computation**, 2ème édition Pearson Education International, 2001.

Plan

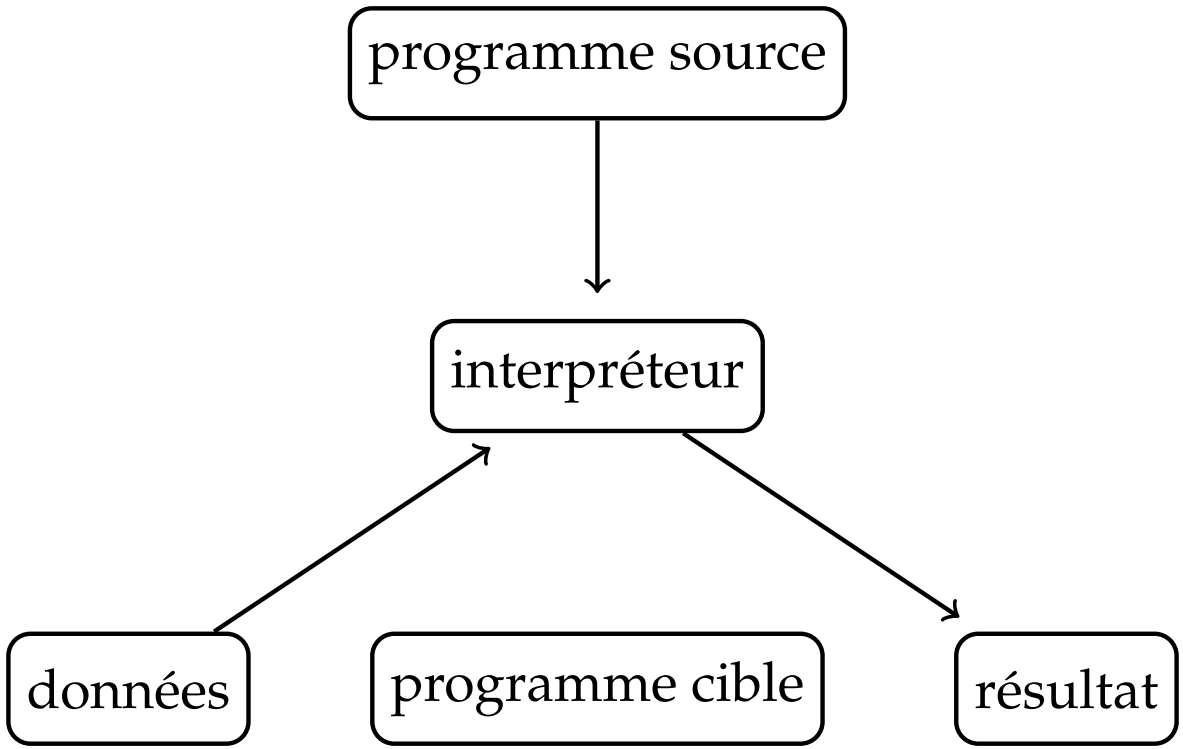
Compilateur

- Un compilateur est un programme
  1. qui lit un autre programme rédigé dans un langage de programmation, appelé langage source
  2. et qui le traduit dans un autre langage, le langage cible.
- Le compilateur signale de plus toute erreur contenue dans le programme source
- Lorsque le programme cible est un programme exécutable, en langage machine, l'utilisateur peut ensuite le faire exécuter afin de traiter des données et de produire des résultats.

Compilateur / interpréteur



Compilateur / interpréteur



Un interpréteur est un programme qui effectue lui-même les opérations spécifiées par le programme source directement sur les données fournies par l'utilisateur.

Exemple

programme source

```
entier $d;

f(entier $a, entier $b)
entier $c, entier $k;
{
 $k = $a + $b;
 retour $k;
}

main()
{
 $d = 7;
 ecrire(f($d, 2) + 1);
}
```

programme cible

```
f:    pop  $t0
      pop  $t1
      push $ra
      add  $t2, $t0, $t1
      sw   $t2, k
      sw   $t2, f
      pop  $ra
      push $t2
      jra
main: li   $t0, 7
      sw   $t0, d
      push $t0
      li   $t1, 2
      push $t2
      jla  f
      pop  $t2
      ...
```

Nécessité d'une analyse syntaxique

- A l'exception de quelques cas (rares), la traduction ne peut être faite "mot à mot"
- Le programme source doit être décomposé en composants pertinents ou constructions du langage source.
- La traduction d'une construction dépend de la position qu'elle occupe au sein du programme.

Décomposition d'une définition de fonction

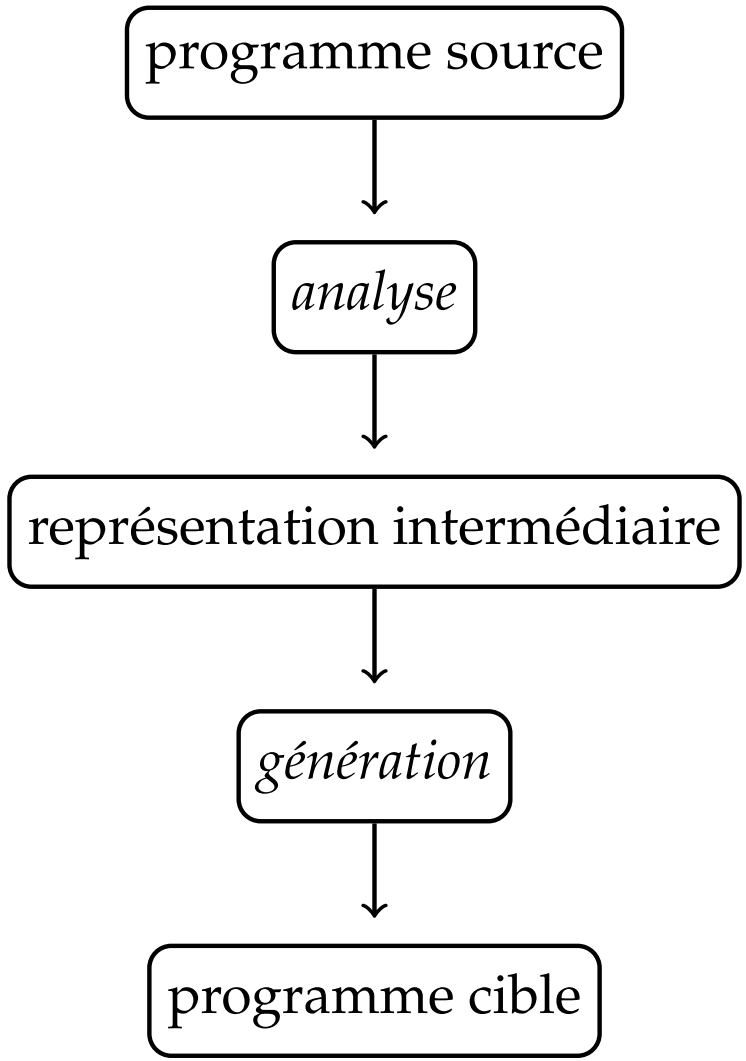
```
entier $d;                          /* variable globale */

f                                   /* nom de la fonction */
(entier $a, entier $b)              /* paramètres de la fonction */
entier $c, entier $k;               /* variables locales */
{                                   /* debut du corps de la fonction */
 $k =                               /* affectation */
    $a + $b;                        /* expression arithmétique */
 retour $k;                         /* valeur de retour */
}                                   /* fin du corps de la fonction */
```

Deux parties d'un compilateur

La traduction du programme source en programme cible se décompose en deux étapes :

- L'analyse, réalisée par la partie frontale du compilateur, qui
  - découpe le programme source en ses constituants ;
  - détecte des erreurs de syntaxe ou de sémantique ;
  - produit une représentation intermédiaire du programme source ;
  - conserve dans une table des symboles diverses informations sur les procédures et variables du programme source.
- La génération, réalisée par la partie finale du compilateur, qui
  - construit le programme cible à partir de la représentation intermédiaire et de la table des symboles

Deux parties d'un compilateur



Nature de la représentation intermédiaire

La conception d'une bonne RI est un compromis :

- Elle doit être raisonnablement facile à produire à partir du programme source.
- Elle doit être raisonnablement facile à traduire vers le langage cible.

Elle doit donc être raisonnablement éloignée (ou raisonnablement proche) du langage source et du langage cible.

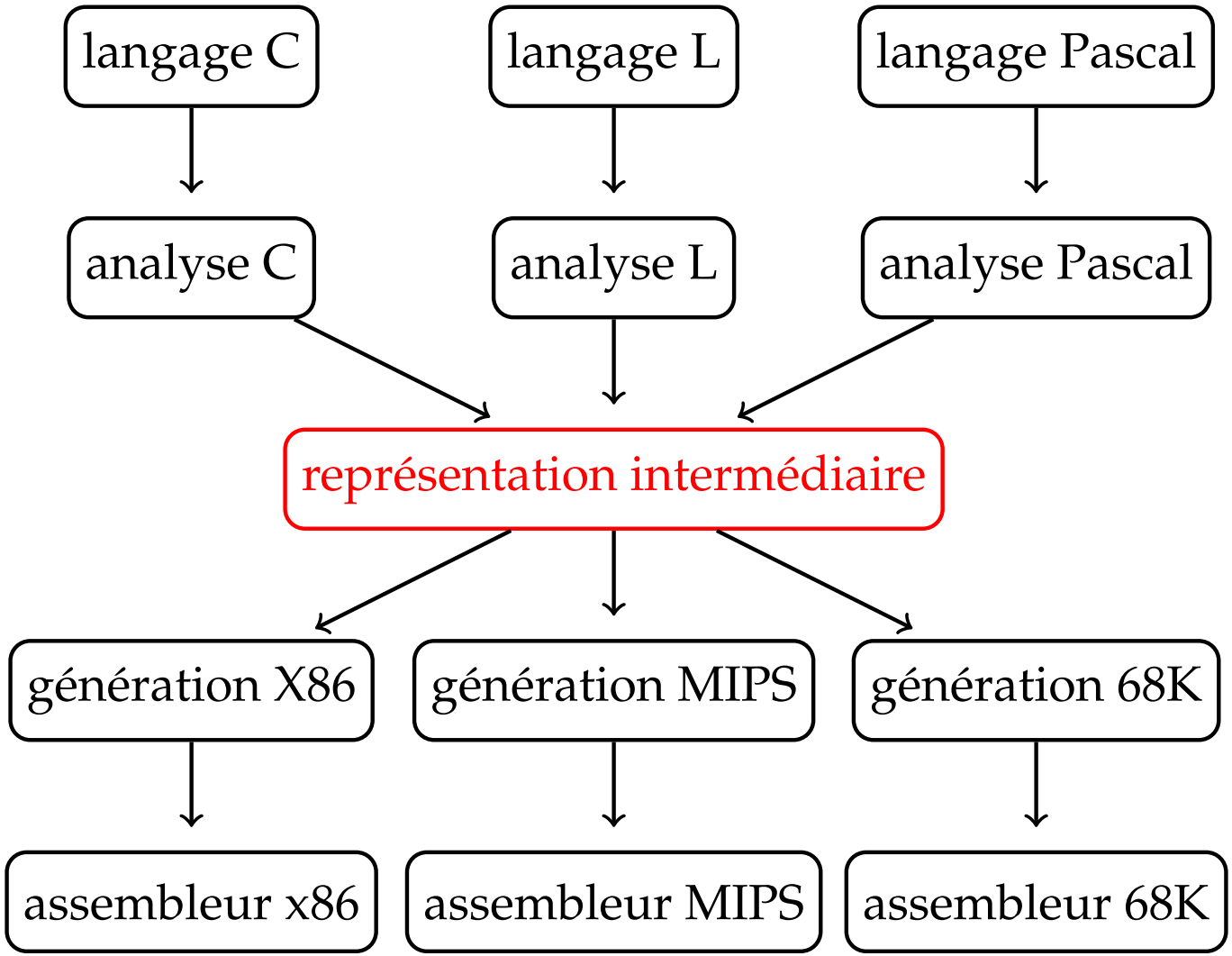
Economie

Une RI judicieusement définie permet de constuire un compilateur pour le langage L et la machine M en combinant :

- un analyseur pour le langage L
- un générateur pour la machine M

Economie : on obtient $m \times n$ compilateurs en écrivant seulement m analyseurs et n générateurs

Portabilité



Syntaxe du langage source

- Le programme source vérifie un certain nombre de contraintes syntaxiques.
- L'ensemble de ces contraintes est appelé grammaire du langage source.
- Si le programme ne respecte pas la grammaire du langage, il est considéré incorrect et le processus de compilation échoue.

Description de la grammaire du langage source

- Littéraire
  - Un programme est une suite de définitions de fonction
  - Une définition de fonction est composée
    - du nom de la fonction suivie de ses arguments
    - suivie de la declaration de ses variables internes
    - suivie d'un bloc d'instructions
  - Une instruction est ...
- Formelle

$$
\begin{array}{rcl}
\text{programme} & \rightarrow & \text{listeDecFonc } '.' \\
\text{listeDecFonc} & \rightarrow & \text{decFonc listeDecFonc} \\
\text{listeDecFonc} & \rightarrow & \\
\text{decFonc} & \rightarrow & \text{ID\_FCT listeParam listeDecVar } ' ;' \text{ instrBloc} \\
 & \ldots &
\end{array}
$$

Grammaires formelles

- Les contraintes syntaxiques sont représentées sous la forme de règles de réécriture.
- La règle $A \rightarrow BC$ nous dit que le symbole A peut se réécrire comme la suite des deux symboles B et C.
- L'ensemble des règles de réécriture constitue la grammaire du langage.
- La grammaire d'un langage L permet de générer tous les programmes corrects écrits en L et seulement ceux-ci

Notations et Terminologie

- Dans la règle $A \to \alpha$
  - A est appelé partie gauche de la règle.
  - α est appelé partie droite de la règle.
- Lorsque plusieurs règles partagent la même partie gauche :

$$A \to \alpha_1 \ , \ A \to \alpha_2 \ , \ldots , \ A \to \alpha_n$$

On les note :

$$A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

Grammaire partielle des expressions arithmétiques

EXPRESSION	$\rightarrow$	EXPRESSION OP2 EXPRESSION
OP2	$\rightarrow$	+ | - | * | /
EXPRESSION	$\rightarrow$	NOMBRE
EXPRESSION	$\rightarrow$	(EXPRESSION)
NOMBRE	$\rightarrow$	CHIFFRE | CHIFFRE NOMBRE
CHIFFRE	$\rightarrow$	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- Les symboles EXPRESSION, OP2, NOMBRE, CHIFFRE sont appelés symboles non terminaux de la grammaire
- Les symboles +, -, *, /, (,), 0, 1 , ..., 9 sont appelés symboles terminaux de la grammaire

Avantages des grammaires formelles

Une grammaire formelle :

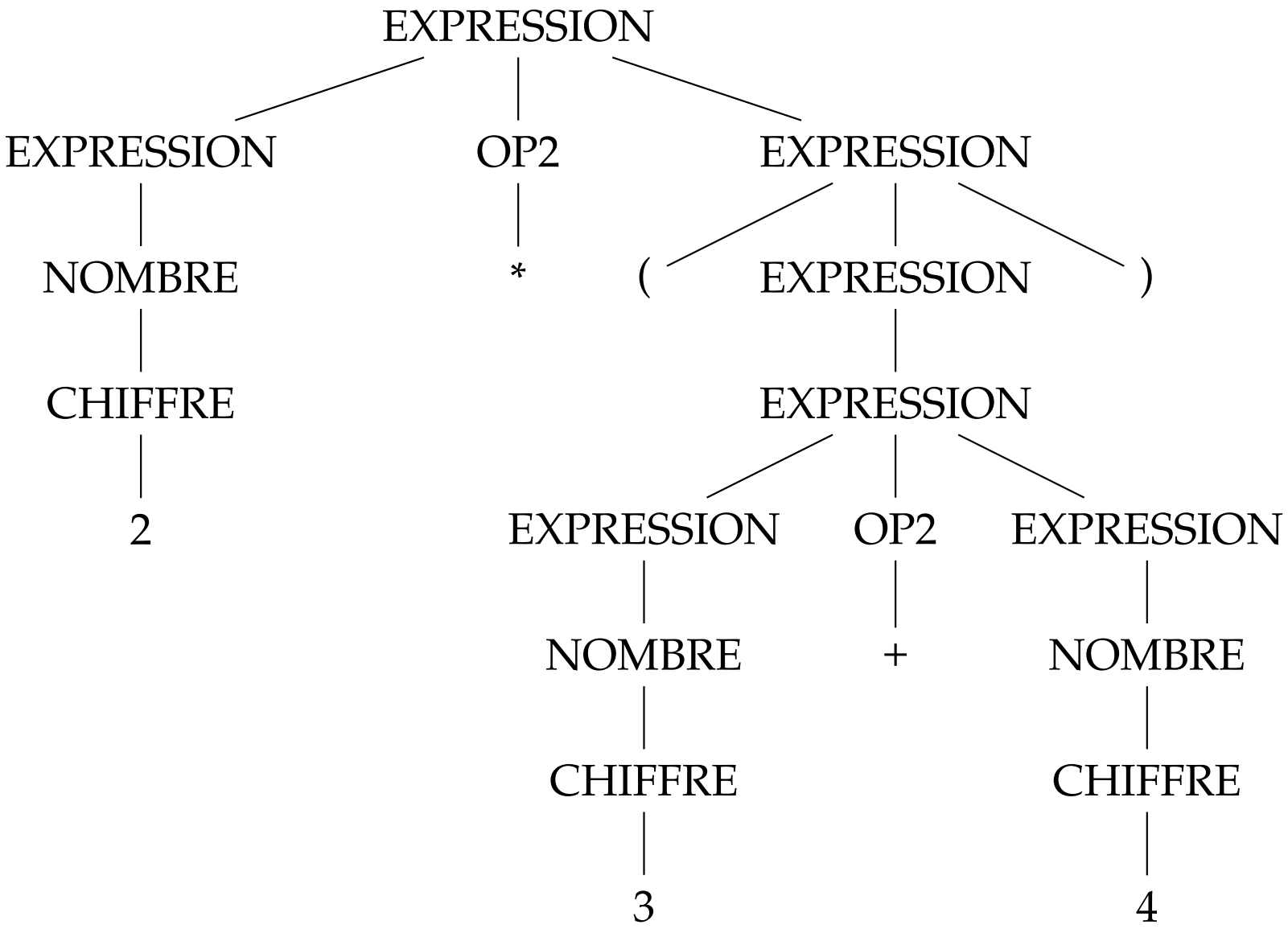
- Pousse le concepteur d'un langage à en décrire la syntaxe de manière exhaustive.
- Permet de répondre automatiquement à la question *mon programme est-il correct ?* à l'aide d'un analyseur syntaxique.
- Fournit, à l'issue de l'analyse, une représentation explicite de l'organisation du programme en constructions (structure syntaxique du programme).
- Cette représentation est utile pour la suite du processus de compilation.

Dérivation d'une expression arithmétique

L'expression arithmétique $2 * (3 + 1)$ est-elle correcte ?

EXPRESSION $\Rightarrow$ EXPRESSION OP2 EXPRESSION
$\Rightarrow$ NOMBRE OP2 EXPRESSION
$\Rightarrow$ CHIFFRE OP2 EXPRESSION
$\Rightarrow$ 2 OP2 EXPRESSION
$\Rightarrow$ 2 * EXPRESSION
$\Rightarrow$ 2 * (EXPRESSION)
$\Rightarrow$ 2 * (EXPRESSION OP2 EXPRESSION)
$\Rightarrow$ 2 * (NOMBRE OP2 EXPRESSION)
$\Rightarrow$ 2 * (CHIFFRE OP2 EXPRESSION)
$\Rightarrow$ 2 * (3 OP2 EXPRESSION)
$\Rightarrow$ 2 * (3 + EXPRESSION)
$\Rightarrow$ 2 * (3 + NOMBRE)
$\Rightarrow$ 2 * (3 + CHIFFRE)
$\Rightarrow$ 2 * (3 + 1)

Arbre de dérivation



Analyse lexicale

- Afin de simplifier la grammaire décrivant un langage, on omet de cette dernière la génération de certaines parties simples du langage.
- Ces dernières sont prises en charge par un analyseur lexical
- L'analyseur lexical traite le programme source et fournit le résultat de son traitement à l'analyseur syntaxique.

Nouvelle grammaire des expressions arithmétiques

Syntaxe	EXPRESSION	→	EXPRESSION OP2 EXPRESSION
	EXPRESSION	→	NOMBRE
	EXPRESSION	→	(EXPRESSION)
Lexique	OP2	→	+ | - | * | /
	NOMBRE	→	CHIFFRE | CHIFFRE NOMBRE
	CHIFFRE	→	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- La nouvelle grammaire omet les détails de la génération d'un NOMBRE et d'un opérateur binaire. Cette partie est à la charge de l'analyseur lexical.
- La frontière entre analyse lexicale et analyse syntaxique est en partie arbitraire.

Analyseur lexical

- Lit le programme source
- Reconnaît des séquences de caractères significatives appelées lexèmes
- Pour chaque lexème, l'analyseur lexical émet un couple

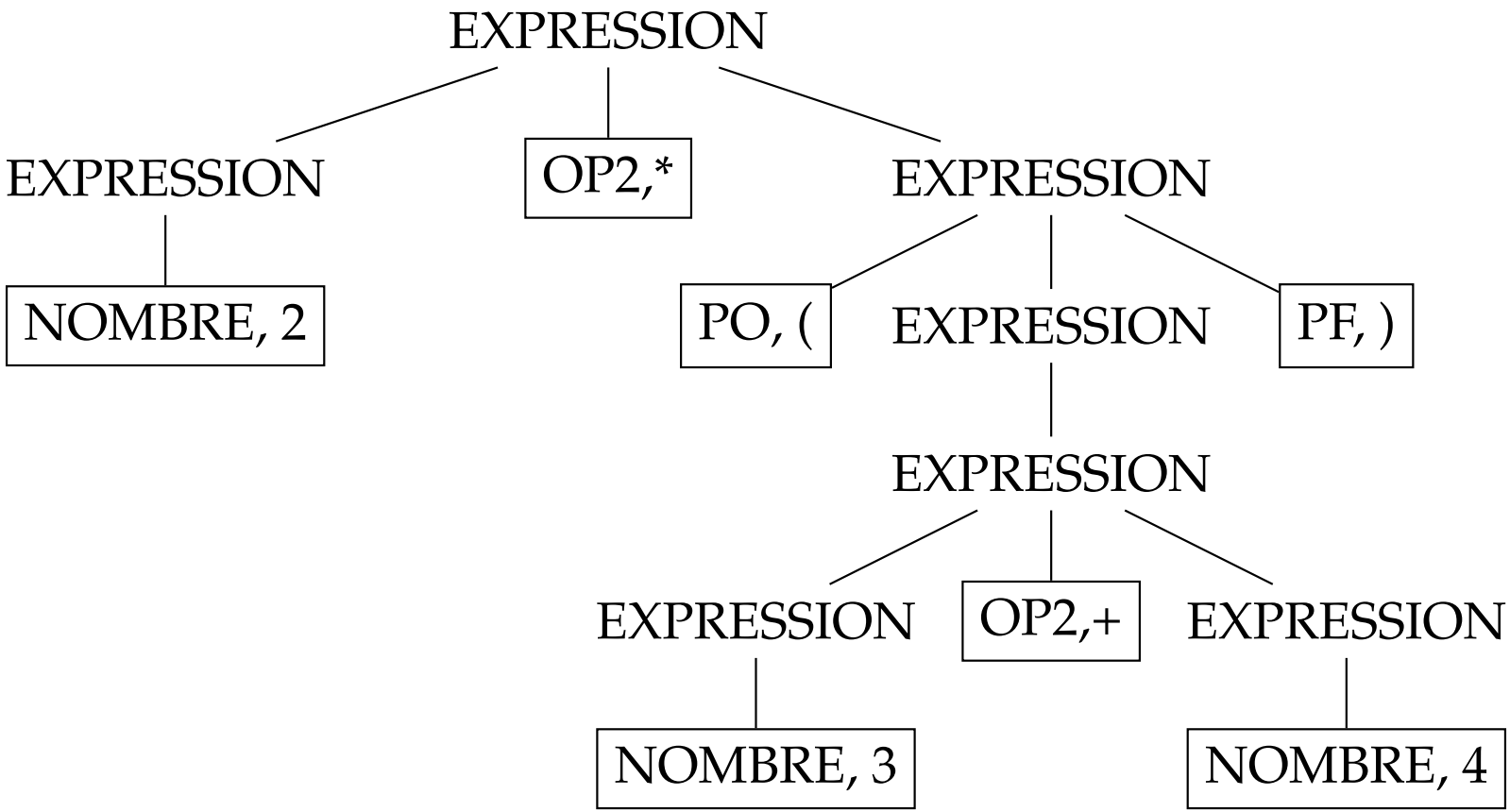
$$(\textit{type du lexème}, \textit{valeur du lexème})$$

- Exemple

$$(\textit{NOMBRE}, 123)$$

- Les types de lexèmes sont des symboles, ils constituent les symboles terminaux de la grammaire du langage.
- Les symboles terminaux de la grammaire (ou types de lexèmes) constituent l'interface entre l'analyseur lexical et l'analyseur syntaxique. Ils doivent être connus des deux.

Analyseur syntaxique plus simple



Traduction dirigée par la syntaxe

La traduction dirigée par la syntaxe est réalisée en attachant des actions sémantiques aux règles de la grammaire.
Elle repose sur deux concepts :

- Les attributs. Un attribut est une quantité quelconque associée à une construction du langage de programmation.
- Exemples :
  - le type d'une expression
  - la valeur d'une expression
  - le nombre d'instructions dans le code généré
- Les constructions étant représentées par les symboles de la grammaire, on associe les attributs à ces derniers.
- Notations : $A.t$ est l'attribut t associé au symbole A.

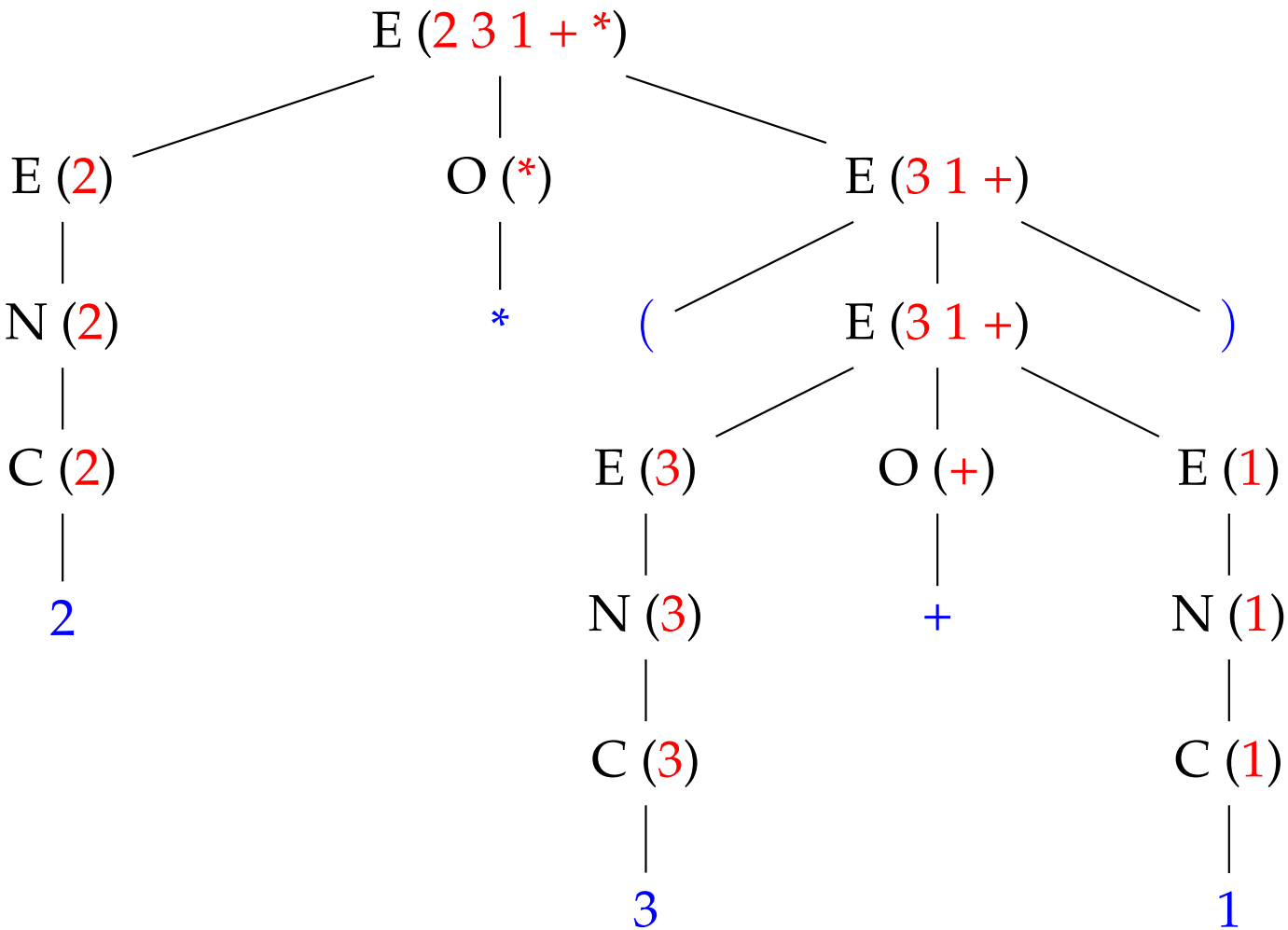
Traduction dirigée par la syntaxe

- Les schémas de traduction sont une notation permettant d'attacher des fragments de programme aux règles de la grammaire.
- Les fragments sont exécutés quand la production est utilisée lors de l'analyse syntaxique.
- Le résultat combiné des exécutions de tous ces fragments, dans l'ordre induit par l'analyse syntaxique, produit la traduction du programme auquel ce processus est appliqué.

Traduction — exemple

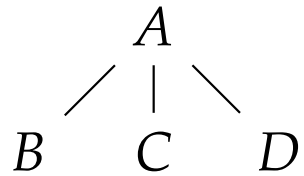
règle			action sémantique		
E	$\rightarrow$	E O E	$E.t$	$=$	$E_1.t \mid\mid E_2.t \mid\mid O.t$
E	$\rightarrow$	(E)	$E.t$	$=$	$E_1.t$
E	$\rightarrow$	N	$E.t$	$=$	$N.t$
O	$\rightarrow$	$+$	$O.t$	$=$	$+$
O	$\rightarrow$	$-$	$O.t$	$=$	$-$
N	$\rightarrow$	C N	$N.t$	$=$	$C.t \mid\mid N_2.t$
N	$\rightarrow$	C	$N.t$	$=$	$C.t$
C	$\rightarrow$	0	$C.t$	$=$	0
C	$\rightarrow$	1	$C.t$	$=$	1
C	$\rightarrow$	2	$C.t$	$=$	2
	$\dots$			$\dots$	
C	$\rightarrow$	9	$C.t$	$=$	9

Traduction — Exemple



Attributs synthétisés

- Un attribut est dit synthétisé si sa valeur au niveau d'un nœud A d'un arbre d'analyse est déterminée par les valeurs de cet attribut au niveau des fils de A et de A lui même.



$$A.t = f(B.t, C.t, D.t)$$

- Les attributs synthétisés peuvent être évalués au cours d'un parcours ascendant de l'arbre de dérivation.
- Un tel parcours peut être effectué simultanément à la construction de l'arbre (lors de l'analyse syntaxique).
- Dans l'exemple, $B.t$, $C.t$ et $D.t$ doivent être calculés avant de calculer $A.t$

Table des symboles

- Elle rassemble toutes les informations utiles concernant les variables et les fonctions ou procédures du programme.
- Pour toute variable, elle garde l'information de :
  - son nom
  - son type
  - sa portée
  - son adresse en mémoire
- Pour toute fonction ou procédure, elle garde l'information de :
  - son nom
  - sa portée
  - le nom et le type de ses arguments, ainsi que leur mode de passage
  - éventuellement le type du résultat qu'elle fournit
- La table des symboles est construite lors de l'analyse syntaxique.

Table de symboles — exemple

programme source

```
entier $d;

f(entier $a, entier $b)
entier $c, entier $k;
{
 $k = $a + $b;
 retour $k;
}

main()
{
 $d = 7;
 ecrire(f($d, 2) + 1);
}
```

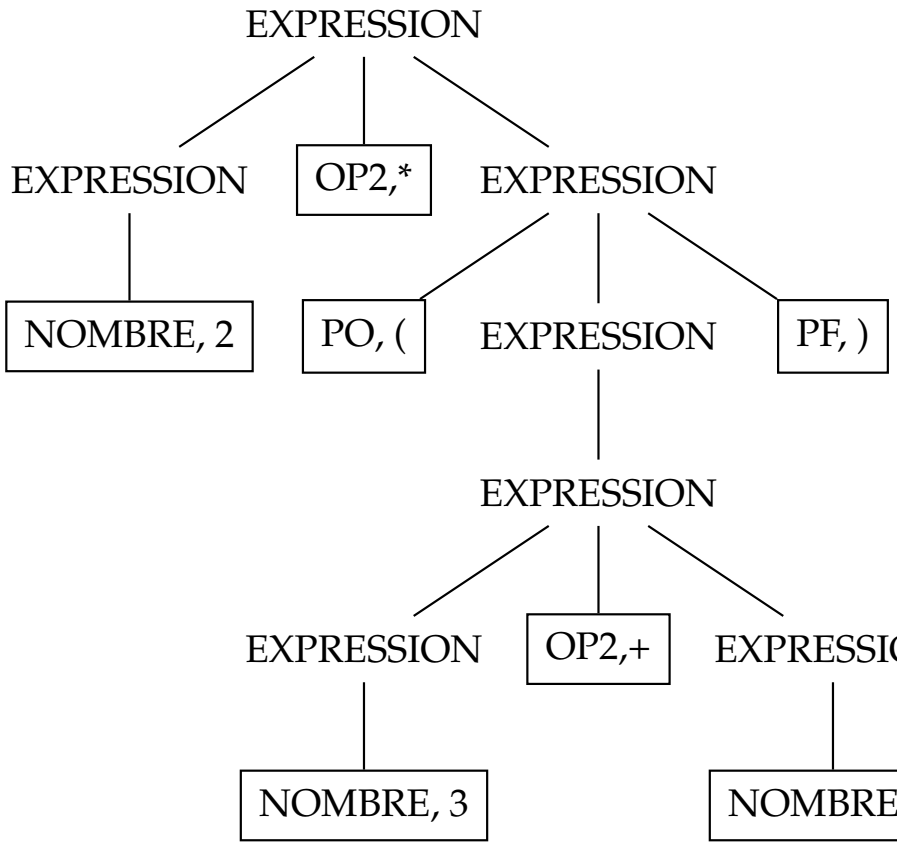
table des symboles

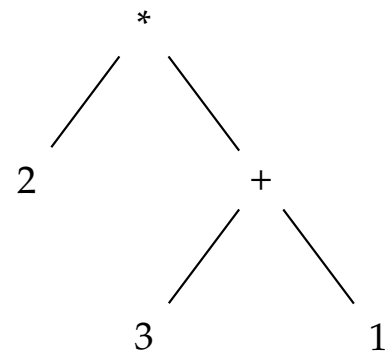
id	classe	type	adr	args
`$d`	globale	entier	1	
`f`	fonct	entier	1	2
`$a`	param	entier		
`$b`	param	entier		
`$c`	locale	entier		
`$k`	locale	entier		
`main`	fonct	entier	11	0

Arbre de dérivation v/s arbre abstrait

- L'arbre de dérivation produit par l'analyse syntaxique possède de nombreux nœuds superflus, qui ne véhiculent pas d'information.
- De plus, la mise au point d'une grammaire nécessite souvent l'introduction de règles dont le seul but est de simplifier l'analyse syntaxique.
- Un arbre abstrait constitue une interface plus naturelle entre l'analyse syntaxique et l'analyse sémantique, elle ne garde de la structure syntaxique que les parties nécessaires à l'analyse sémantique et à la production de code.
- L'arbre abstrait est construit lors de l'analyse syntaxique, en associant à toute règle de grammaire une action sémantique.

Arbre de dérivation v/s arbre abstrait





Analyse sémantique

L'analyse sémantique utilise l'arbre abstrait, ainsi que la table de symboles afin d'effectuer un certain nombre de contrôles sémantiques, parmi lesquels :

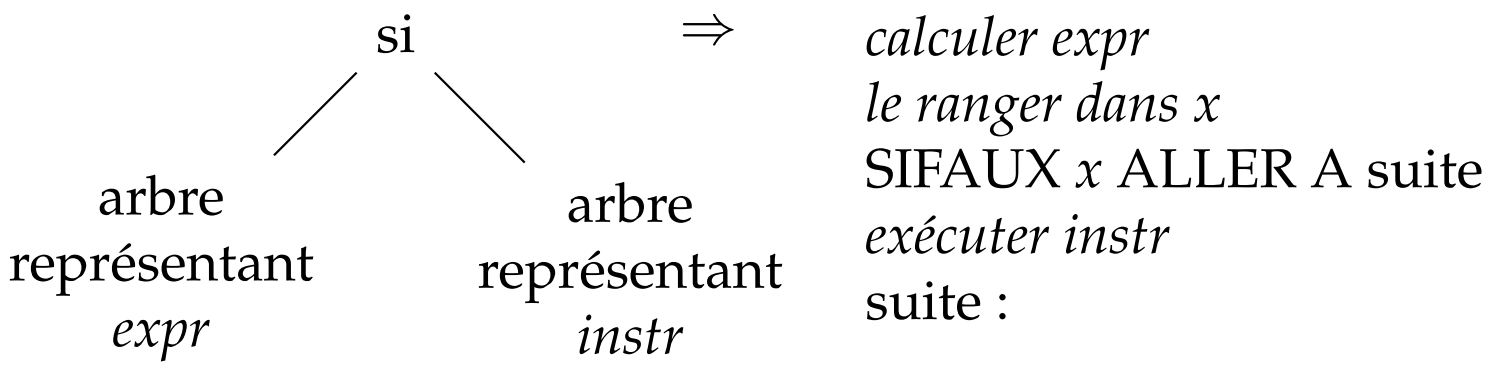
- vérifier que les variables utilisées ont bien été déclarées.
- le contrôle de type : le compilateur vérifie que les opérandes d'un opérateurs possèdent bien le bon type.
- conversions automatiques de types.

Représentation intermédiaire

- La production de code consiste à produire une séquence d'instructions sémantiquement équivalente au programme source qui pourra être interprétée par une machine donnée.
- La production de code est effectuée lors du parcours de la Représentation intermédiaire (RI) construit à l'issue le l'étape d'analyse.
- Plusieurs choix sont possibles pour la RI :
  - l'arbre abstrait
  - une séquence d'instructions élémentaires
  - ...

Exemple

Production de code lors du parcours d'un nœud associé à une instruction **si** *expr* **alors** *instr*.



Code à 3 adresses

Avant d'écrire tout-à-fait le code, on peut passer par une version simplifiée qui consiste principalement à représenter le programme sous la forme d'une séquence d'instructions simples

```
tantque $i < $n-1 faire
{
  $i = $i * 2 + 1;
}
ecrire(i)
```

```
2: load $i
3: load $n
4: loadimm 1
5: moins 3, 4
6: inf 2, 5
7: jsifaux 6, 15
8: load $i
9: loadimm 2
10: fois 8, 9
11: loadimm 1
12: plus 10, 11
13: store 12, $i
14: jump 2
15: load $i
16: ecrire 15
```

Sources

- A.Aho, M.Lam, R.Sethi et J.Ullman,
  Compilateurs : principes, techniques et outils.
  2ème édition. Pearson Education, 2007
- A.Appel,
  Modern compiler implementation in C.
  Cambridge University Press, 1998
- Henri Garreta,
  Polycopié du cours de compilation.
  `http://henri.garreta.perso.luminy.univmed.fr/Polys/PolyCompil.pdf`

Projet

- Construction en langage C d'un compilateur.
- Le langage source appelé L est un langage impératif simple.
- Le langage cible est l'assembleur MIPS. Il est interprété par une machine virtuelle appelée `spim`.

Le langage *L*

- Proche du langage C ou de Pascal
- quelques caractéristiques :

Types : Le langage *L* connaît deux types de variables :

- Un type simple : le type `entier`.
- Un type dérivé : les tableaux d'entiers.

Variables : Les noms des variables commencent par dollar ($)

Opérateurs : Le langage *L* connaît les opérateurs suivants :

- arithmétiques : `+`, `-`, `*`, `/`
- comparaison : `<`, `=`
- logiques : `&` (et), `|` (ou), `!` (non)

Le langage *L*

Instructions : Le langage *L* connaît les instructions suivantes :

- Instruction vide
  `;`
- Bloc d'instructions, délimité par des accolades
  `{ ... }`
- Affectation [2]
  `$a = $b + 1;`
- Instructions de contrôle
  `si` *expression* `alors { ... }`
  `si` *expression* `alors { ... } sinon { ... }`
  `tantque` *expression* `faire { ... }`
- Retour de fonction
  `retour` *expression* `;`
- Instruction d'appel à fonction simple
  `fonction(` *liste d'expressions* `);`

2. Contrairement à C, une affectation n'est pas une expression $\Longrightarrow$ on ne peut écrire `$a = $b = 4`

Le langage *L*

Sous-programmes : un programme *L* est une suite de sous-programmes, dont `main`

- Ce sont des fonctions à résultat entier.
- Le passage se fait par valeur.
- Les fonctions possèdent des variables locales.
- Une fonction ne peut pas être déclarée à l'intérieur d'une autre.
- On peut ignorer le résultat rendu par une fonction.

Procédures pré-définies : Les entrées-sorties de valeurs entières se font à l'aide de deux fonctions prédéfinies :

- `$a = lire();`
- `ecrire($a);`

Exemple

```
f(entier $a, entier $b)  # déclaration d'une fonction à deux arguments
entier $c, entier $k;    # déclaration de deux variables locales
{                        # début d'un bloc d'instruction
  $k = $a + $b;          # affectation et expression arithmétique
  retour $k;             # valeur de retour de la fonction
}                        # fin du bloc d'instruction

main()                   # point d'entrée dans le programme
entier $d;
{
  $d = f($d, 2);         # affectation et appel de fonction
  ecrire($d + 1);        # appel de la fonction prédéfinie ecrire
}
```

Assembleur MIPS

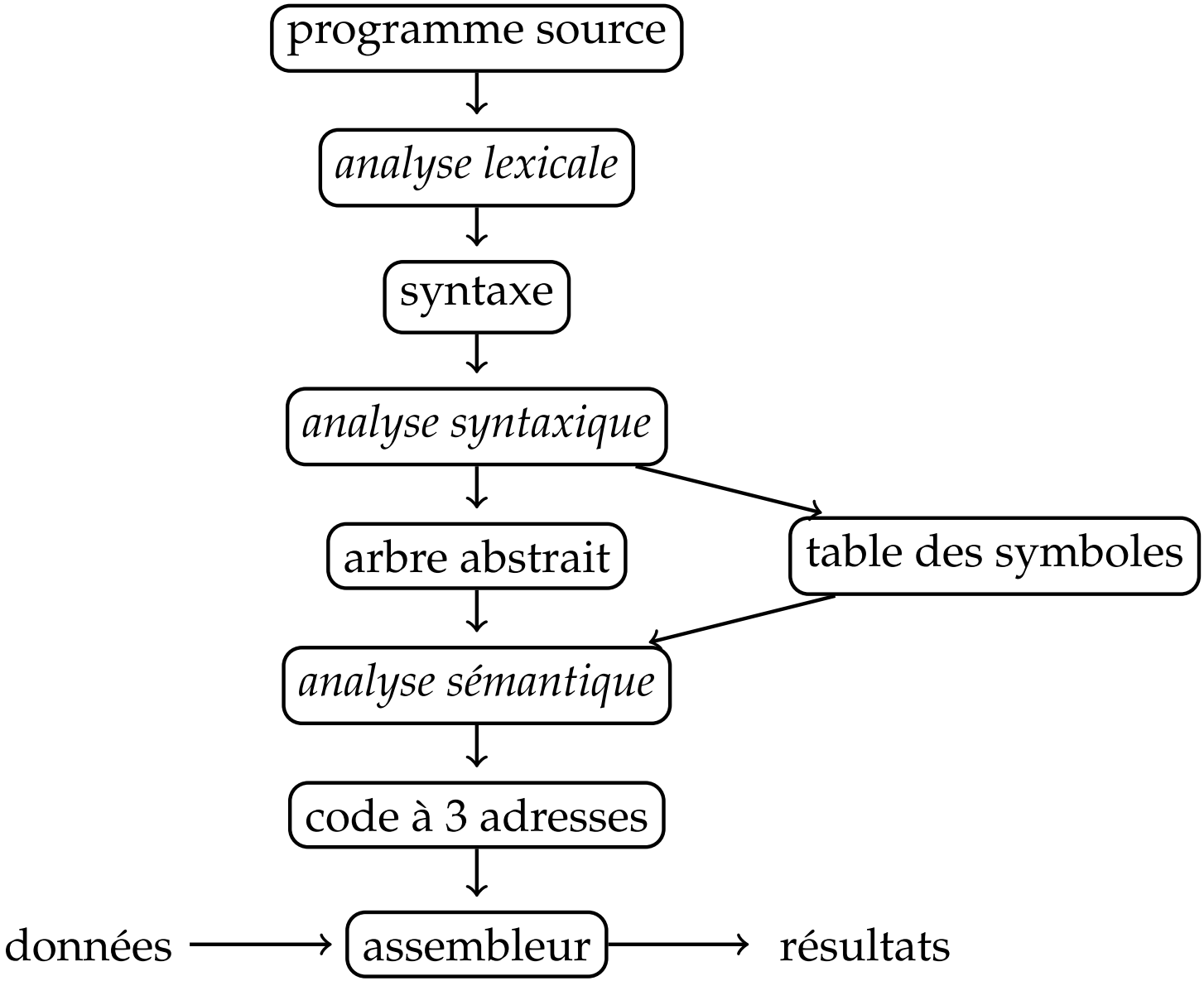
Langage utilisé par le processeur MIPS. On utilisera la machine virtuelle `spim`.

- Mémoire séparée en trois parties :
  - l'espace global (variables)
  - la zone de code,
  - la pile.
- En plus, registres utilisés dans la plupart des instructions.
  - registres courants : `$t0-$t9`,
  - pointeur de sommet de pile : `$sp`,
  - adresse de retour de saut : `$ra` ...

MIPS — exemples

`lw $t0, k`	charge la variable à l'adresse `k` dans `$t0`
`li $t1, 8`	charge la valeur 8 dans `$t1`
`add $t2, $t0, $t1`	addition : `$t2 :=$t0 + $t1`
`jump label`	saute à l'adresse `label` dans le code

Structure du compilateur



Étapes

TP	durée	intitulé
1	1	analyseur lexical
2-3	2	analyseur syntaxique
4	1	production de l'arbre abstrait
5	1	évaluation partielle
6	1	analyse sémantique et table des symboles
7-8	2	production de code MIPS
9	1	appels à fonction, optimisations
10	1	évaluation finale