Introduction à la Compilation

Alexis Nasr
Carlos Ramisch
Manon Scholivet
Franck Dary

Compilation – L3 Informatique
Département Informatique et Interactions
Aix Marseille Université

Infos pratiques

- 9 cours, 10 séances de TD et 10 séances de TP
- Emploi du temps :
  - Consulter l'ENT
- Évaluation session 1 :
  - Partiel (30%)
  - Projet (30%)
  - Examen final (40%)
- Évaluation session 2 :
  - Projet (30%)
  - Examen final (70%)
- Page web du cours :
  `http://pageperso.lif.univ-mrs.fr/~alexis.nasr`

Bibliographie

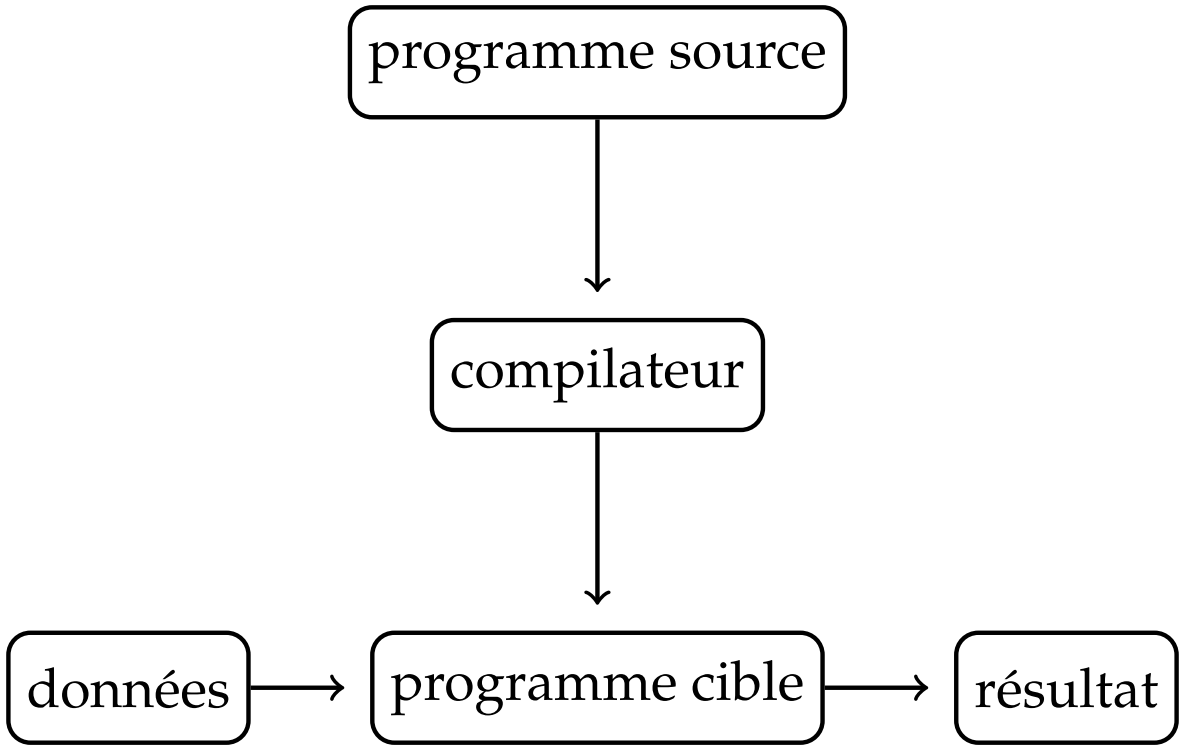
- Alfred Aho, Monica Lam, Ravi Sethi et Jeffrey Ullman **Compilateurs principes, techniques et outils**, 2ème édition. Pearson Education, 2007
- Andrew Appel **Modern compiler implementation in JAVA**, 2nd edition, Cambridge University Press, 2002.
- John Hopcroft, Rajeev Motwani, Jeffrey Ullman **Introduction to Automata Theory, Languages and Computation**, 2ème édition Pearson Education International, 2001.
- Michael Sipser **Introduction to the Theory of Computation** PWS Publishing Company, 1997.

Plan

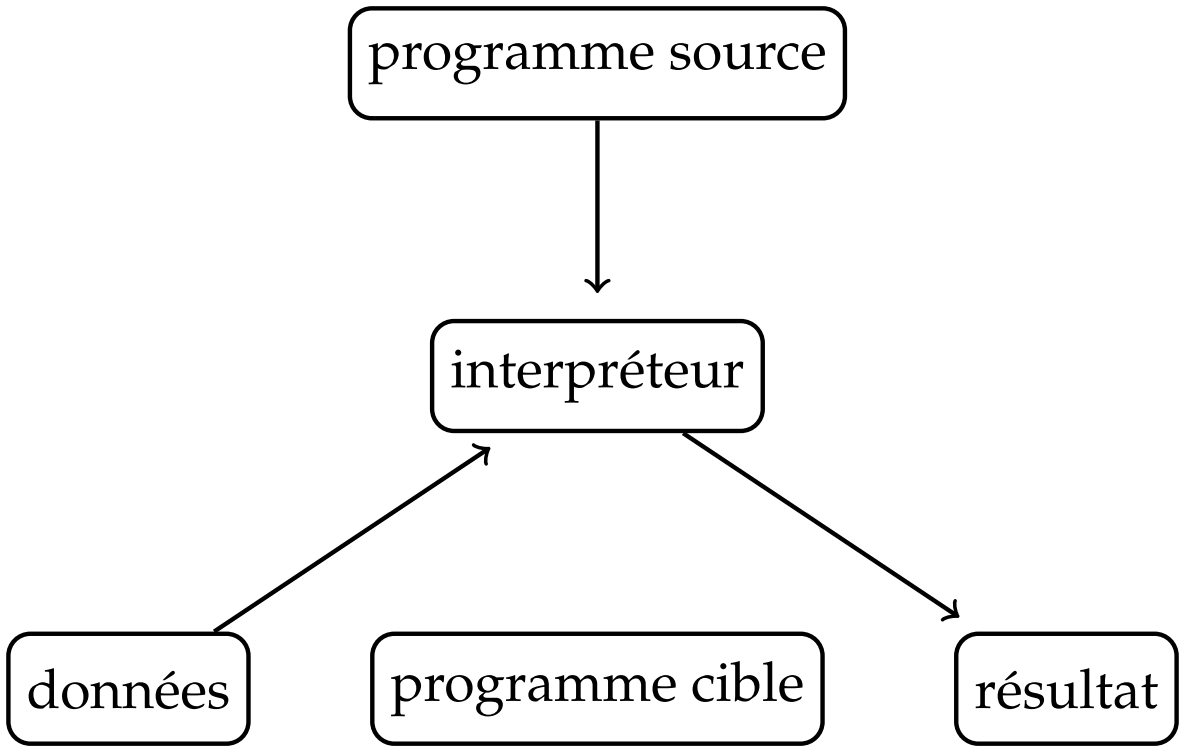
Compilateur

- 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:
 push ebp
 mov  ebp, esp
 sub  esp, 8
 mov  ebx, [ebp + 12]
 push ebx
 mov  ebx, [ebp + 8]
 push ebx
 ...
main:
 push ebp
 mov  ebp, esp
 sub  esp, 8
 push 7
 pop  ebx
 mov  [d], ebx
 ...
```

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

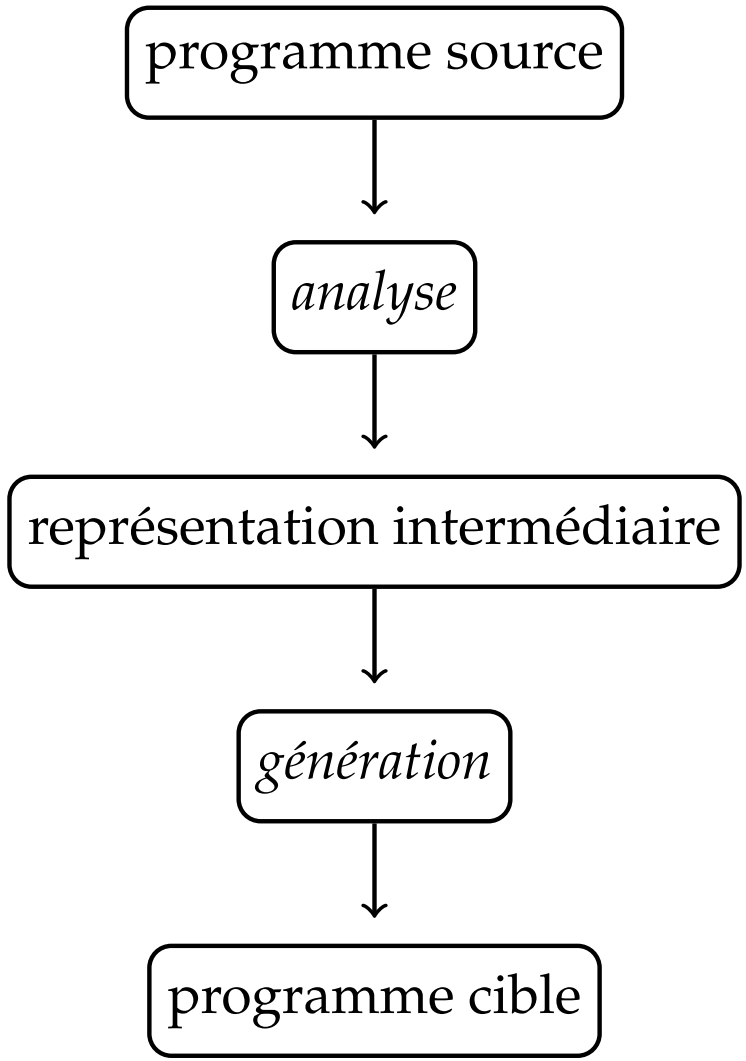
```
entier d;                          /* variable globale */

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.

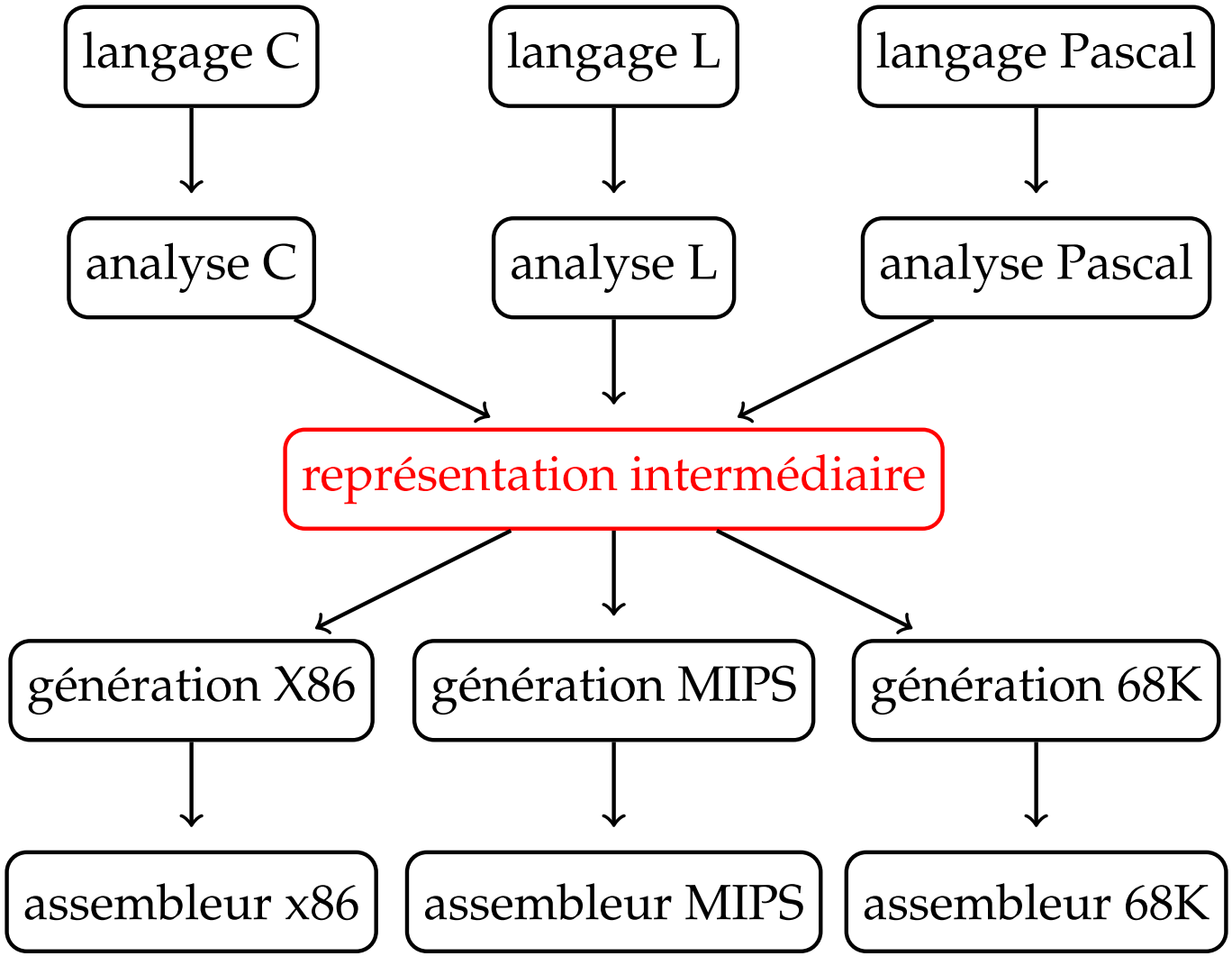
Economie

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

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

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 :

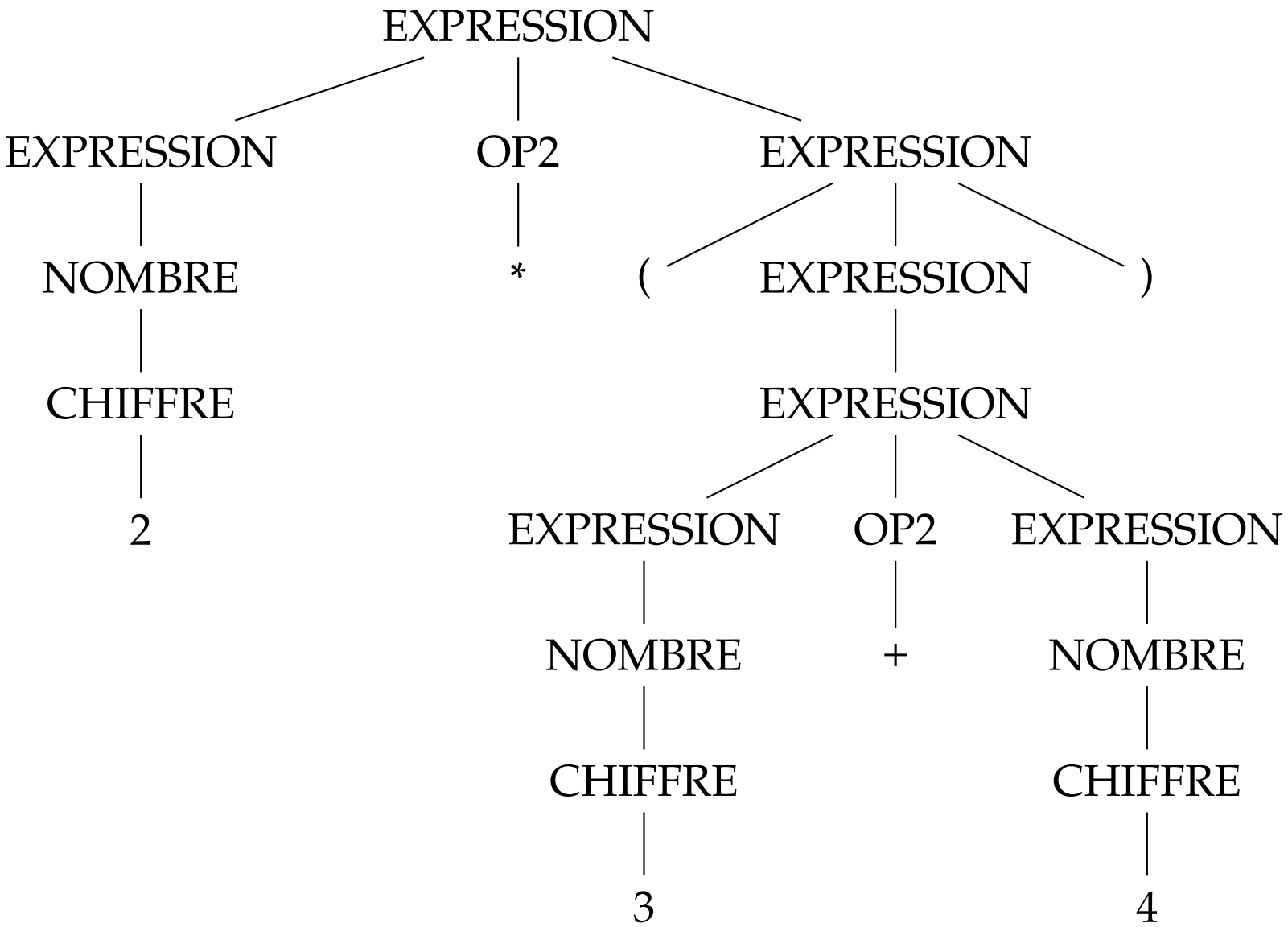
- Pousse le concepteur d'un langage à en décrire la syntaxe de manière exhaustive.
- Permet de répondre automatiquement à la question *mon programme est-il correct ?* à l'aide d'un analyseur syntaxique.
- Fournit, à l'issue de l'analyse, une représentation explicite de l'organisation du programme en 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	$\rightarrow$	EXPRESSION OP2 EXPRESSION
	EXPRESSION	$\rightarrow$	NOMBRE
	EXPRESSION	$\rightarrow$	(EXPRESSION)
Lexique	OP2	$\rightarrow$	+ | - | * | /
	NOMBRE	$\rightarrow$	CHIFFRE | CHIFFRE NOMBRE
	CHIFFRE	$\rightarrow$	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

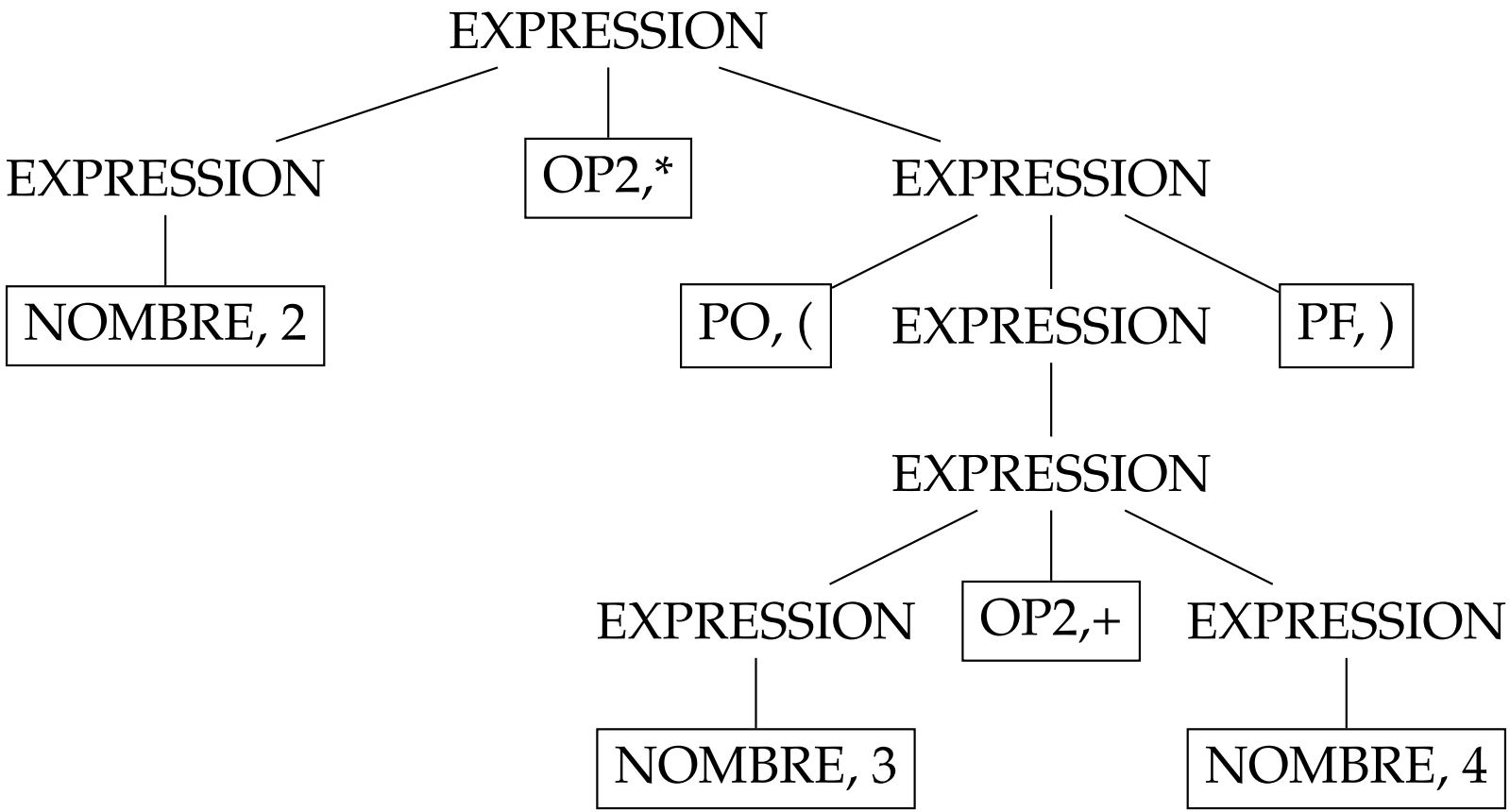
$$(\textit{type du lexème}, \textit{valeur du lexème})$$

- 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 consiste à attacher des actions aux règles de la grammaire.
- Ces actions peuvent être exécutées :
  - lors de l'analyse syntaxique
  - ou après, lors d'un parcours de l'arbre de dérivation.
- Le résultat de ces exécutions constitue une traduction du programme analysé.
- La traduction dirigée par la syntaxe repose sur deux concepts :
  - Les attributs
  - Les actions sémantiques

Attributs

- Les attributs sont des valeur quelconques associées aux constructions 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.
- Une grammaire dont les symboles ont été enrichis d'attributs est appelée grammaire attribuée.

Actions sémantiques

- Les actions sémantiques permettent de calculer la valeur des attributs.
- Elles se présentent généralement sous la forme d'affectations dont la partie gauche est un attribut.
- Lorsqu'une règle possède plusieurs occurrences d'un même symbole, on les enrichit, dans les actions sémantique, d'un indice afin de les distinguer.
- Exemple :

règle	action sémantique
$E \rightarrow E + E$	$E.t = E_1.t + E_2.t$
$E \rightarrow 5$	$E.t = 5$

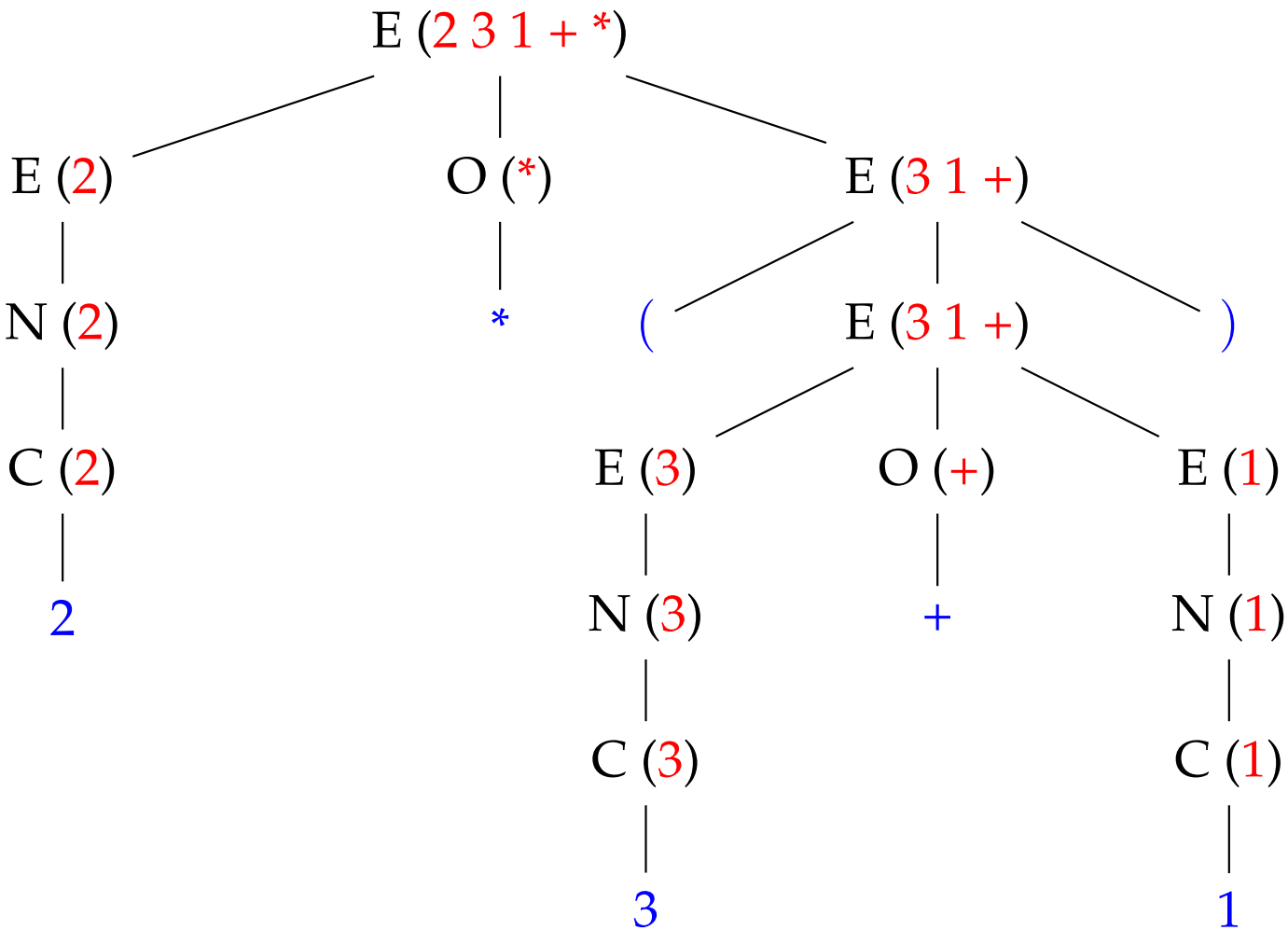
- Conventions :
  - le symbole de la partie gauche n'a pas d'indice,
  - lorsque la partie droite possède plusieurs occurrences d'un même symbole, la première occurrence a pour indice 1, la seconde 2 ...

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

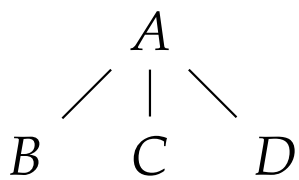
- $\mid\mid$ dénote la concaténation
- “Mini” compilateur notation infixe $\rightarrow$ notation postfixe

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.



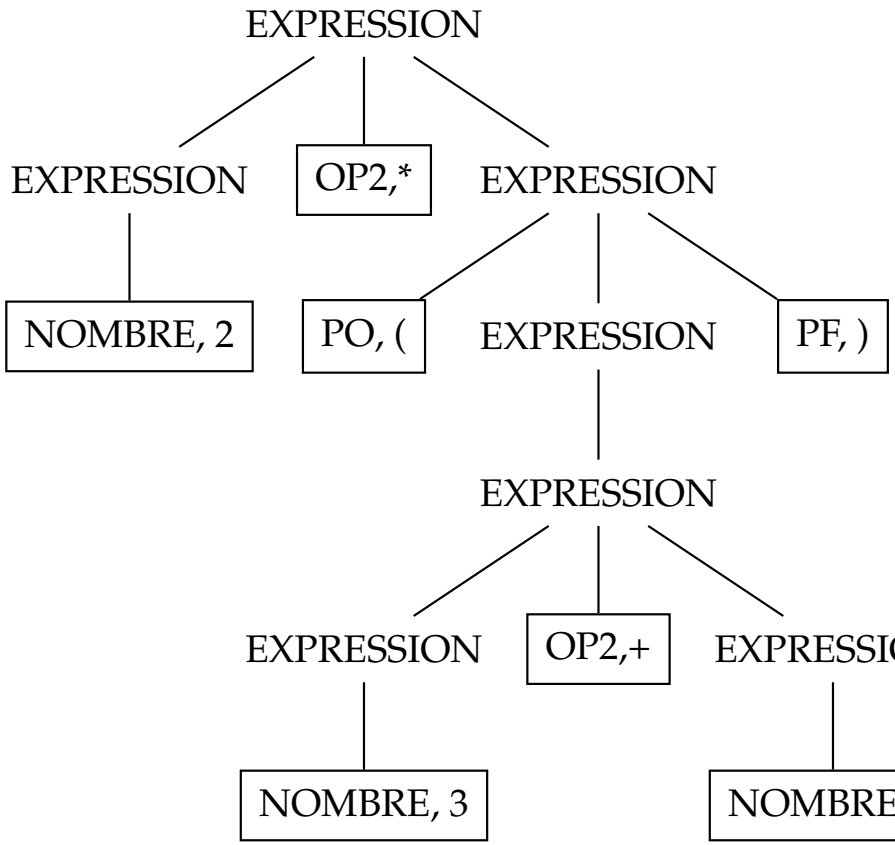
$$A.t = f(B.t, C.t, D.t)$$

- Les attributs synthétisés peuvent être évalués au cours d'un parcours ascendant de l'arbre de dérivation.
- Dans l'exemple, $B.t$, $C.t$ et $D.t$ doivent être calculés avant de calculer $A.t$

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 au reste des traitements.
- L'arbre abstrait est construit lors de l'analyse syntaxique ou lors du parcours de l'arbre de dérivation, grâce à la traduction dirigée par la syntaxe.

Arbre abstrait



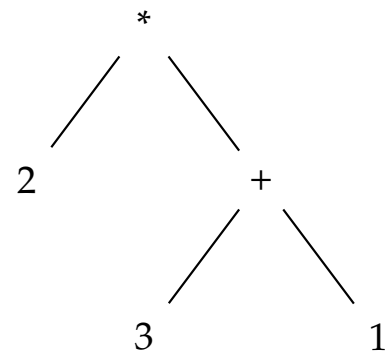


Table des symboles

- 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 peut être construite lors de l'analyse syntaxique ou lors du parcours de l'arbre de dérivation ou de l'arbre abstrait.
- Il peut y avoir plusieurs tables des symboles, une pour chaque portée.

Table de symboles — exemple

programme source

```
entier T[10],
entier v;



f(entier a, entier b)
entier c, entier k;
{}


main()
entier d;
{}
```

tables des symboles

TABLE GLOBALE			
T	VAR	10	0
v	VAR	1	10
f	FCT	2	
main	FCT	0	

TABLE : f			
a	PARAM	1	0
b	PARAM	1	1
c	VAR	1	0
k	VAR	1	1

TABLE : main			
d	VAR	1	0

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 :

- vérification que les variables utilisées ont bien été déclarées.
- contrôle de type : les opérandes d'une opération possèdent bien le bon type.
- conversions automatiques de types.

Code trois adresses

- La production de code consiste à produire une séquence d'instructions sémantiquement équivalente au programme source qui pourra être interprétée par une machine donnée.
- Chaque machine possède ses spécificités (jeu d'instructions, nombre de registre, conventions d'appel ...)
- Afin de s'abstraire de ces détails, on définit un jeu d'instruction abstrait appelé, appelé code trois adresses.
- Le code trois adresses constitue la représentation intermédiaires.

Code trois adresses

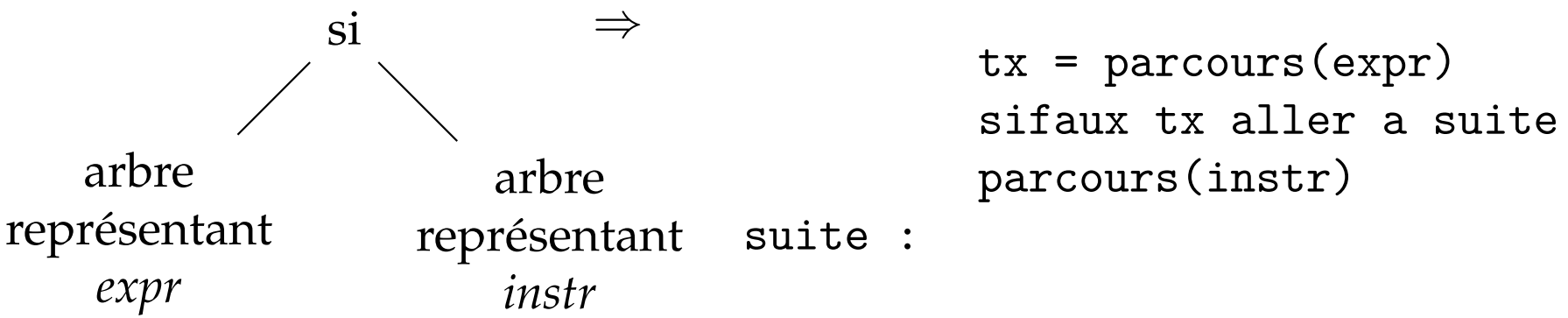
- Toute instruction du code trois adresses comporte au plus un opérateur dans sa partie droite.
- Les expressions complexes doivent être décomposées en expressions plus simples
- Exemple : L'expression `3 + 4 * 2` doit être décomposé de la manière suivante :

```
t1 = 4 * 2
t2 = 3 + t1
```

- Les variables `t1` et `t2` sont créées par le compilateur, il peut y en avoir un nombre arbitraire.

Production du code trois adresses

- La production du code trois adresses est réalisée lors du parcours de l'arbre abstrait
- Exemple : production de code lors du parcours d'un nœud associé à une instruction **si** *expr* **alors** *instr*.



Exemple

```
main()
entier i;
{
i = 0;
tantque i < 10 faire
{
  i = i * 2 + 1;
}
ecrire(i);
}
```

```
main   fbegin
       i = 0
l0     t0 = -1
       if i < 10 goto l2
       t0 = 0
l2     if t0 = 0 goto l1
       t1 = i * 2
       t2 = t1 + 1
       i = t2
       goto l0
l1     write i
       fend
```

Génération de code

- Le code machine est composé d'instructions pour un processeur
- Il y a autant de langages machine que des processeurs
- Le code machine est généré à partir de la représentation intermédiaire
- La génération de code consiste à :
  1. Traduire les instructions du code trois adresses en instructions du processeur
  2. Allouer la mémoire nécesssaire pour le programme
  3. Ranger les temporaires dans des registres ou dans la mémoire
  4. Gérer les appels de fonctions grâce à la pile

Exemple : NASM

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:
 push ebp
 mov  ebp, esp
 sub  esp, 8
 mov  ebx, [ebp + 12]
 push ebx
 mov  ebx, [ebp + 8]
 push ebx
 ...
main:
 push ebp
 mov  ebp, esp
 sub  esp, 8
 push 7
 pop  ebx
 mov  [d], ebx
 ...
```

Sources

- A.Aho, M.Lam, R.Sethi et J.Ullman,
  Compilateurs : principes, techniques et outils.
  2ème édition. Pearson Education, 2007
- Andrew Appel,
  Modern compiler implementation in JAVA, 2nd edition
  Cambridge University Press, 2002.

Projet

- Construction en langage JAVA d'un compilateur.
- Le langage source appelé L est un langage impératif simple.
- Le langage cible est l'assembleur NASM. Il génère des binaires executables pour des processeurs de la famille `x86`.

Le langage L

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

Types : Le langage L connaît deux types :

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

Variables : Les variables doivent être déclarées et sont typées.

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 :

- Bloc d'instructions, délimité par des accolades
  `{ ... }`
- Affectation [1]
  `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* `);`

1. 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 NASM

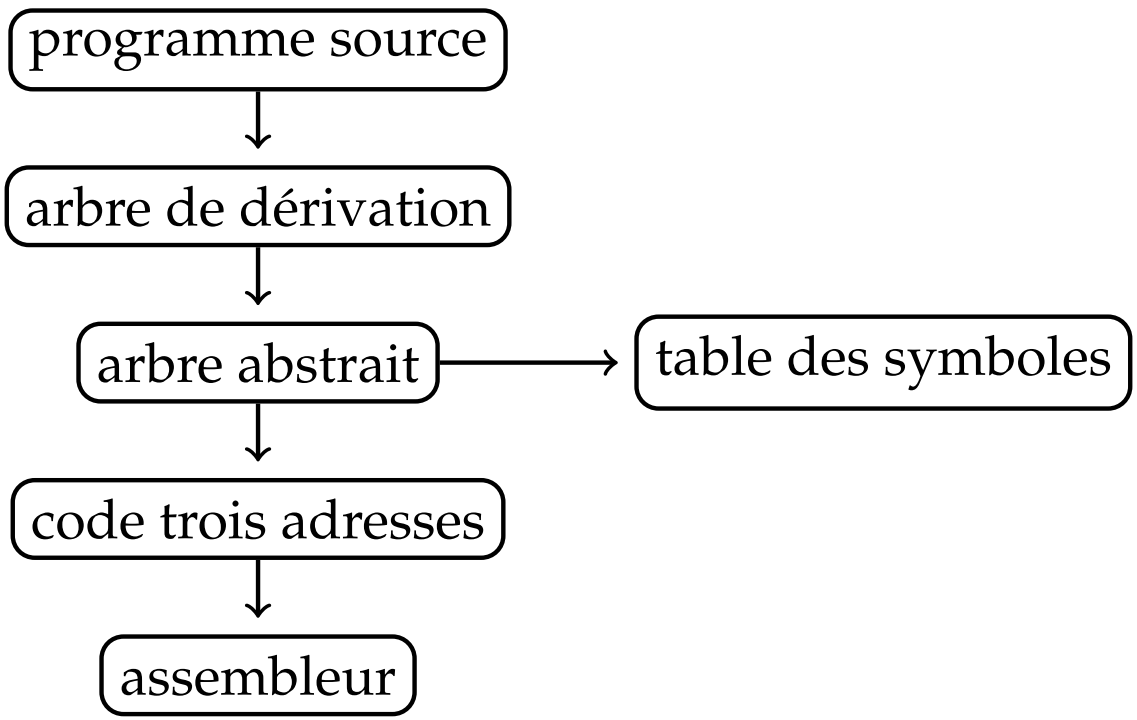
Assembleur pour les processeurs X86.

- 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 : `eax-edx`,
  - pointeur de sommet de pile : `exp` ...

NASM — exemples

`mov eax [k]`	charge le contenu de l'adresse `k` dans `eax`
`mov ebx 1`	charge la valeur `1` dans `ebx`
`add eax, ebx`	addition : `eax = eax + ebx`
`jmp label`	saute à l'adresse label dans le code

Etapes de la compilation



TP

TP	durée	intitulé
1	1	analyseur lexical & syntaxique
2	1	production de l'arbre abstrait
3	1	analyse sémantique et table des symboles
4	2	production du code 3 adresses
5	1	génération de code NASM
6	1	solution du graphe de flot
7	1	allocation de registres
8	1	intégration
9	1	évaluation finale