Notes sur la compilation

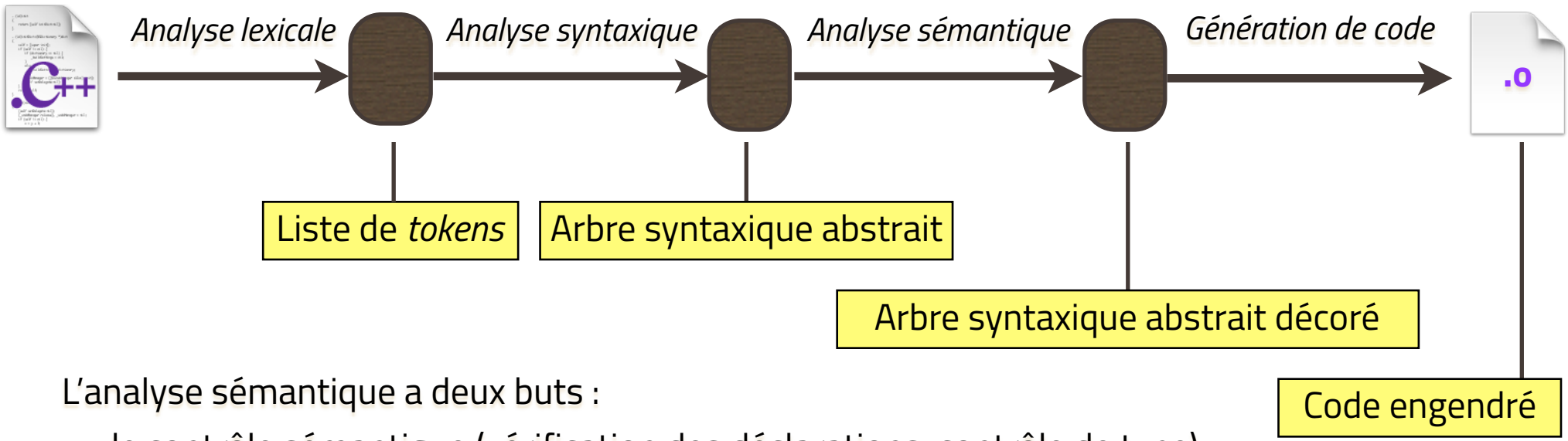
Option INFO — TLANG

Pierre Molinaro

Décembre 2019

CENTRALE NANTES

Structure d'un compilateur



L'analyse sémantique a deux buts :

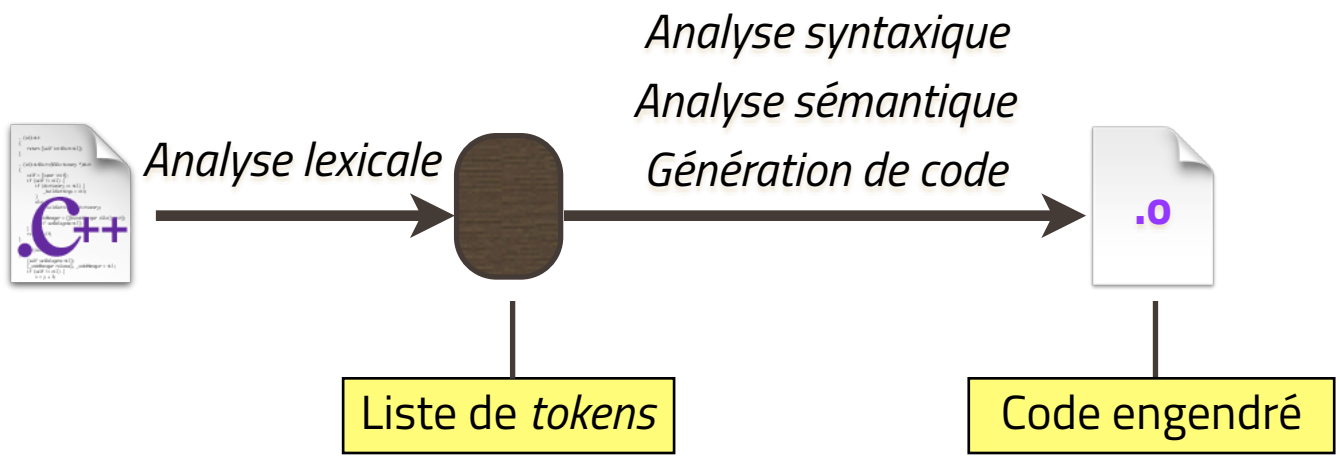
- le contrôle sémantique (vérification des déclarations, contrôle de type) ;
- l'ajout d'informations sémantiques pour guider la génération de code.

Elle est réalisée par un ou plusieurs parcours et / ou transformations de l'arbre syntaxique abstrait.

CENTRALE NANTES

Structure d'un compilateur très simple

Pour des langages très simples, il est possible d'effectuer l'analyse sémantique et la génération de code au fil de l'analyse syntaxique.



Comment écrire un compilateur ?

Les choix de conception d'un langage ont des répercussions importantes sur l'organisation de son compilateur. Dans cette partie, nous allons voir deux points :

- « *Compilation séparée ou tout compiler à chaque fois ?* » ;
- « *Fragile Base Class Problem* » ;
- ordre des déclarations : « *déclaration avant utilisation, ou ordre libre ?* ».

CENTRALE NANTES

Analyse syntaxique

CENTRALE NANTES

Rôle de l'analyse syntaxique

Le rôle de l'analyse syntaxique est de :

- vérifier que le texte source est correct syntaxiquement ;
- construire l'*arbre syntaxique abstrait* (présenté en détail plus loin).

L'arbre syntaxique abstrait est une structure de données qui contient l'information pertinente du texte source.

Liste de déclarations

Très souvent, l'arbre syntaxique abstrait contient une liste de déclarations.

Cette approche simplifie l'écriture de l'analyse sémantique, car cette peut être facilement réordonnée.

En C++, une déclaration peut-être :

- une déclaration de classe ;
- une déclaration de structure ;
- une fonction ;
- le prototype d'une fonction ;
- une variable globale ;
- une constante globale ;
- ...
-

CENTRALE NANTES

Exemple C++

```
#include<iostream>

using namespace std;

struct point {
    double x,y;
};

int main() {
    …
}
```

Attention, en C / C++, ceci n'est pas une déclaration mais un ordre destiné au préprocesseur : le compilateur C / C++ ne voit jamais les directives.

Exemple *Piccolo*

Piccolo est un compilateur pour un assembleur structuré pour les micro-contrôleurs de la famille PIC18.

```
pic18 blink_led "18F448" :

configuration {
  ...
}

ram accessram {
  ...
}

noreturn routine main bank : requires 0 {
 ...
}

end
```

CENTRALE NANTES

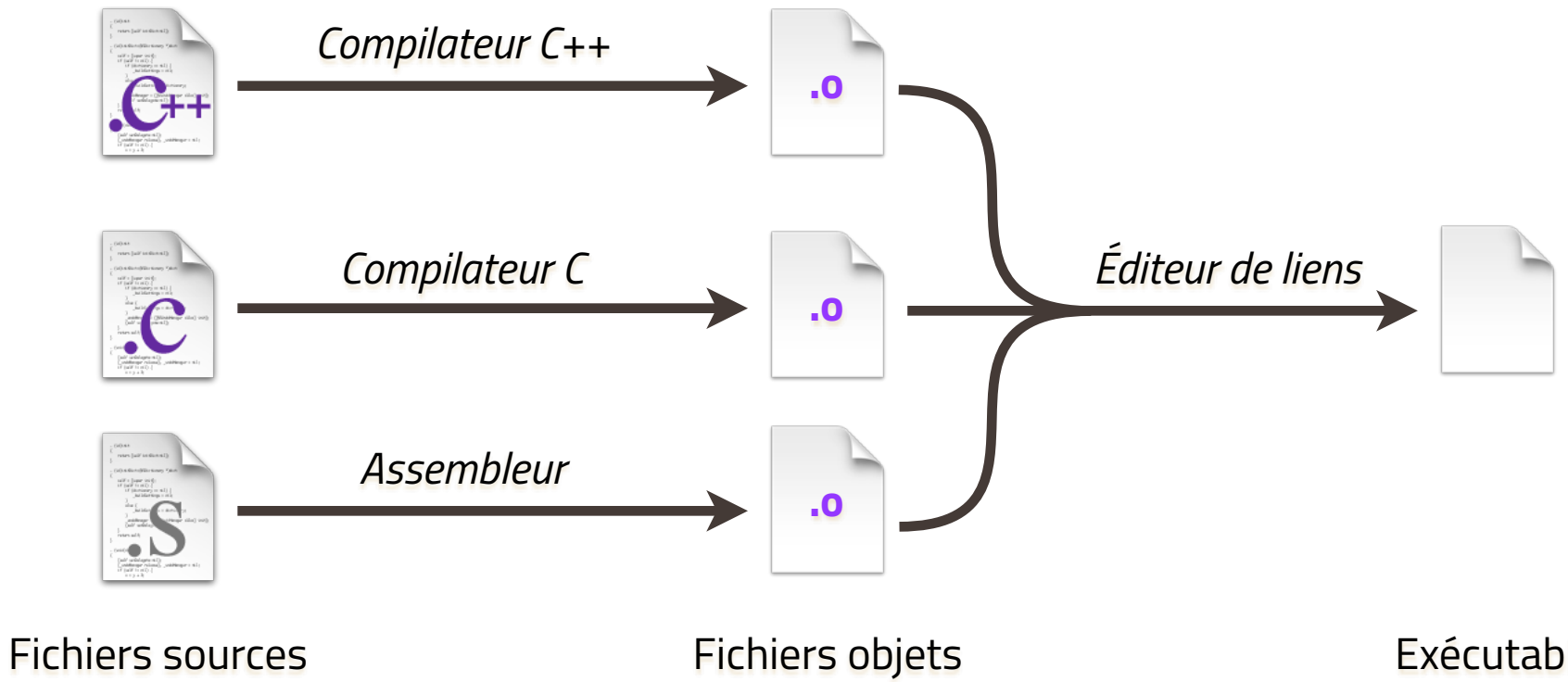
Compilation séparée
ou
tout compiler à chaque fois ?

CENTRALE NANTES

Compilation séparée

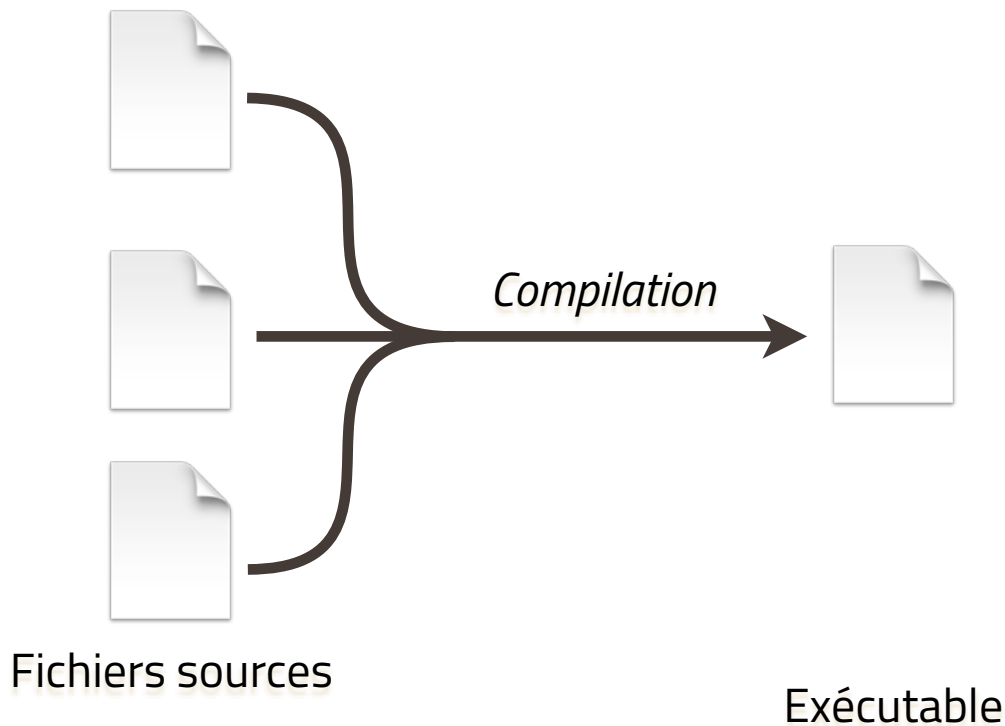
C'est le choix de la plupart des compilateurs C / C++ :

- ne recompile que les sources qui ont changé, et les autres sources qui en dépendent ;
- nécessité de définir un format *objet*, et une *édition des liens*.



Tout compiler à chaque fois

À chaque compilation, tous les fichiers sources sont pris en compte, qu'ils aient été modifiés ou non.



En pratique, que choisir ?

Tout recompiler à chaque fois est envisageable :

- tant que la durée de compilation totale est acceptable ;
- si la notion de *librairie compilée* n'existe pas.

Par exemple :

Piccolo, un compilateur pour un assembleur structuré pour les micro-contrôleurs de la famille PIC18. ROM flash : maximum 128 kio -> max 65536 instructions assembleur, tout compiler à chaque fois : moins d'une seconde ;

GALGAS, générateur de compilateur (utilisé en TP dans ce cours) ; la description de GALGAS en GALGAS : 54 000 lignes sources, 10 700 lignes de templates, le compilateur engendre du texte C++. Durée de compilation GALGAS : environ 5 s.

Fragile Base Class Problem

CENTRALE NANTES

Application Binary Interface

Application Binary Interface (http://en.wikipedia.org/wiki/Application_binary_interface) : ensemble des règles qui assurent l'interopérabilité des différentes chaînes de développement.

Plus précisément, pour chaque langage :

- la représentation interne des types de données,
- leur alignement,
- les conventions d'appel de fonctions, de routines,
- dans un langage objet, comment la liaison dynamique est implémentée, comment l'objet courant est représenté,
- …

Pour les générateurs de code :

- le format des codes objet ;
- le format des librairies ;

Ces règles sont primordiales pour assurer (entre autres) :

- l'interface entre l'assembleur et les autres langages ;
- l'appel des fonctions systèmes.

CENTRALE NANTES

Fragile Base Class Problem

Ce problème s'est révélé crucial lors de la conception de Be OS :

- un logiciel est compilé avec une version des librairies dynamiques du système ;
- une nouvelle version du système est publiée, avec une nouvelle version de ces librairies dynamiques ;
- le logiciel peut-il être exécuté sur la nouvelle version du système ?

C'est-à-dire si le logiciel compilé avec l'ancienne version des librairies système peut s'exécuter avec la nouvelle version des librairies système, et ce, sans être recompilé.

Liens :

https://en.wikipedia.org/wiki/Fragile_binary_interface_problem

https://web.archive.org/web/20130306104745/http://2f.ru/holy-wars/fbc.html

Un exemple (1/2)

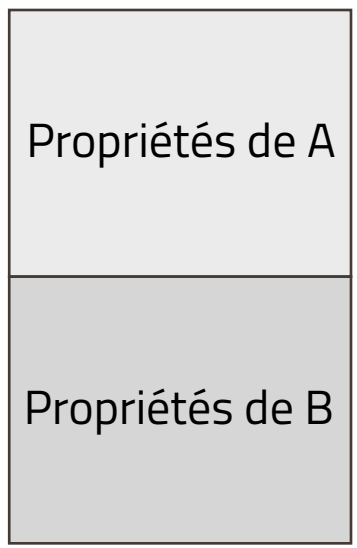
La classe **A** est définie dans une librairie système ; la classe **B**, qui hérite de **A**, est définie dans le logiciel.

```
class A {
  propriétés de A
} ;
```

```
class B : A {
  propriétés de B
} ;
```

Instance de A

Instance de B



Les propriétés de **B** sont ajoutées à la suite de celles de **A**.

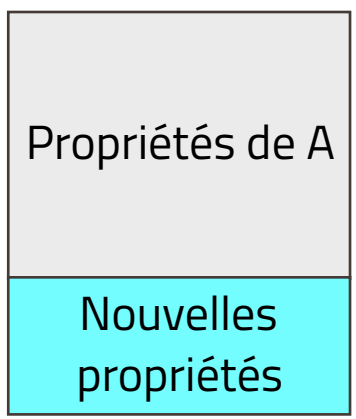
Un exemple (2/2)

Dans la nouvelle version du système, des propriétés sont ajoutées à la classe **A** .

```
class A {
  propriétés de A
  nouvelles propriétés
} ;
```

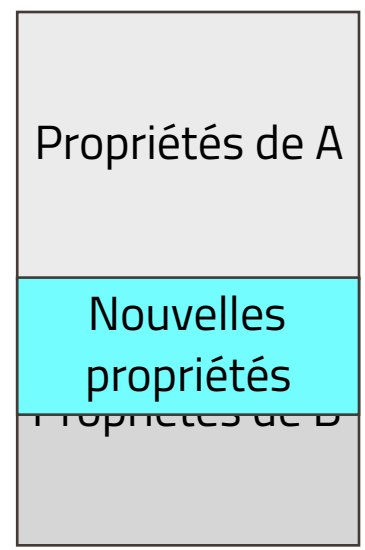
```
class B : A {
  propriétés de B
} ;
```

Instance de A



Si la classe **B** n'est pas recompilée avec la nouvelle version de la classe **A**, ses propriétés sont à la même adresse que les nouvelles propriétés de **A**.

Instance de B



Lien :

https://web.archive.org/web/20130306104745/http://2f.ru/holy-wars/fbc.html

Ordre des déclarations

Ordre des déclarations

Quelque soit le choix *compilation séparée* ou *tout compiler à chaque fois*, il y a un choix de conception du langage qui conditionne l'écriture du compilateur :

- « *déclaration avant utilisation* » ;
- ordre libre des déclarations.

CENTRALE NANTES

Langage obéissant à « *déclaration avant utilisation* »

C'est le cas du langage C et du C++ (avec quelques exceptions).

Par exemple :

- une classe doit être déclarée après une autre classe dont elle hérite ;
- une fonction ne peut être appelée que si elle a été définie ;
- …

```
class A { … } ;

class B : A { … } ;

void fonction ( … ) { … }

void autreFonction ( … } {
  …
  fonction ( … ) ;
  …
}
```

CENTRALE NANTES

Langage obéissant à « *déclaration avant utilisation* » *Avantages*

L'analyseur sémantique est plus simple à écrire : quand l'analyseur atteint un identificateur, il sait exactement comment il est défini.

Si l'identificateur n'est associé à aucune définition, c'est une erreur.

Un exemple typique est le graphe d'héritage des classes : il ne doit pas comporter de circularité. Obéir à « *déclaration avant utilisation* » garantit l'absence de circularité.

De même, le compilateur peut facilement détecter si une structure contient un champ dont le type est la structure elle-même :

```
struct TypeStructure {
  TypeStructure champ ; // Erreur !
} TypeStructure ;
```

Il suffit d'insérer dans la table le type structure **après** l'analyse de tous ces champs.

CENTRALE NANTES

Langage obéissant à « *déclaration avant utilisation* »
Inconvénients

La règle « *déclaration avant utilisation* » est trop contraignante dans certains cas, comme par exemple si l'on veut décrire deux routines récursives qui s'appellent l'une l'autre :

```
void fonctionA ( … ) {
  …
  fonctionB ( … ) ;
  …
}

void fonctionB ( … ) {
  …
  fonctionA ( … ) ;
  …
}
```

CENTRALE NANTES

Langage obéissant à « *déclaration avant utilisation* » *Inconvénients*

En C et C++, il faut *prédéclarer* la seconde fonction avec un prototype :

```
void fonctionB ( … ) ; // Prototype

void fonctionA ( … ) {
  …
  fonctionB ( … ) ;
  …
}

void fonctionB ( … ) {
  …
  fonctionA ( … ) ;
  …
}
```

CENTRALE NANTES

Langage obéissant à « *déclaration avant utilisation* »
Inconvénients

La situation se complique lorsque le projet C / C++ contient plusieurs fichiers source : il faut alors écrire des fichiers d'*en-tête* (headers), qui contiennent les déclarations de classes, de fonctions, …

CENTRALE NANTES

Langage obéissant à « *ordre libre des déclarations* »

L'intérêt est évident pour l'utilisateur : il est beaucoup plus simple d'écrire son programme, puisqu'il ne doit respecter aucun ordre imposé.

Cependant, il faut que le compilateur vérifie certaines propriétés :

- pas d'héritage circulaire ;
- pas de circularité dans les propriétés de structure ;
- …

Par ailleurs, il est indispensable que l'ordre *déclaration avant utilisation* soit établi lors de l'analyse sémantique : cela simplifie l'écriture de l'analyse sémantique.

Stratégie de compilation

Pour un langage obéissant à « *ordre libre des déclarations* », voici une stratégie de compilation :

① Un texte source est une liste de déclarations. Une déclaration peut être une classe, une fonction, une structure, une énumération, …

② Analyse syntaxique de tous les fichiers source. Le but de l'analyse syntaxique est de fournir un *arbre syntaxique abstrait*, ici une liste dont chaque élément est une structure de données qui contient l'information de la déclaration analysée. Les listes obtenues pour chaque fichier sources sont concatenées pour former une seule liste.

③ Pour chaque déclaration, on établit ses dépendances : une classe dépend de sa super classe, une structure des types de ses propriétés, … On construit ainsi un graphe.

④ Pour que le graphe construit soit valide, il faut qu'il soit *acyclique*, c'est-à-dire sans circularité. On effectue donc un *tri topologique* :

- si il réussit, on obtient une liste ordonnée de déclarations respectant les dépendances ;
- si il échoue, il y a des circularités.

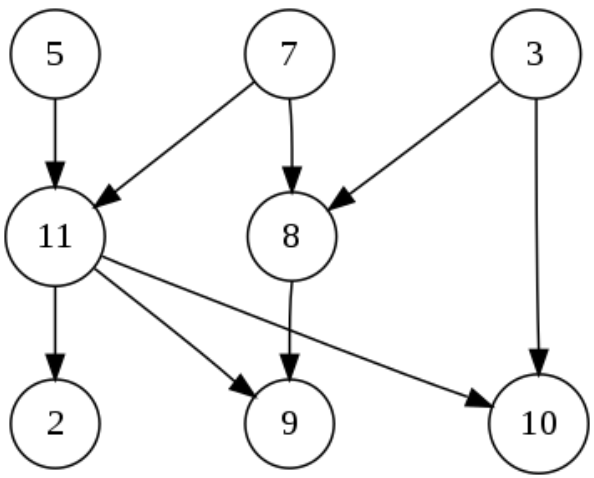
⑤ Parcours de la liste ordonnée pour connaître toutes les entités déclarées ;

⑥ Parcours de la liste ordonnée pour effectuer l'analyse sémantique de chaque déclaration.

Tri topologique

Tri topologique : https://en.wikipedia.org/wiki/Topological_sorting.

Obtenir un ordre compatible avec les contraintes de dépendances.



attention : définition de flèche

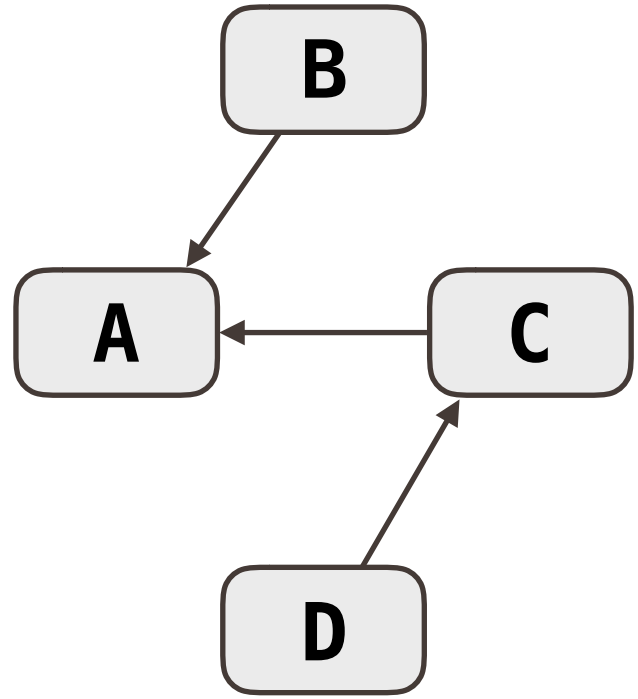
The graph shown to the left has many valid topological sorts, including:

- 5, 7, 3, 11, 8, 2, 9, 10 (visual left-to-right, top-to-bottom)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
- 5, 7, 3, 8, 11, 10, 9, 2 (fewest edges first)
- 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
- 5, 7, 11, 2, 3, 8, 9, 10 (attempting top-to-bottom, left-to-right)
- 3, 7, 8, 5, 11, 10, 2, 9 (arbitrary)

Pour l'analyse sémantique, peu importe l'ordre obtenu, du moment qu'il respecte le tri topologique.

Exemple

```
class B : A { … }

class A { … }

class D : C { … }

class C : A { … }
```



Ordres possibles :

- A B C D ;
- A C B D ;
- A C D B

Arbre Syntaxique Abstrait

CENTRALE NANTES

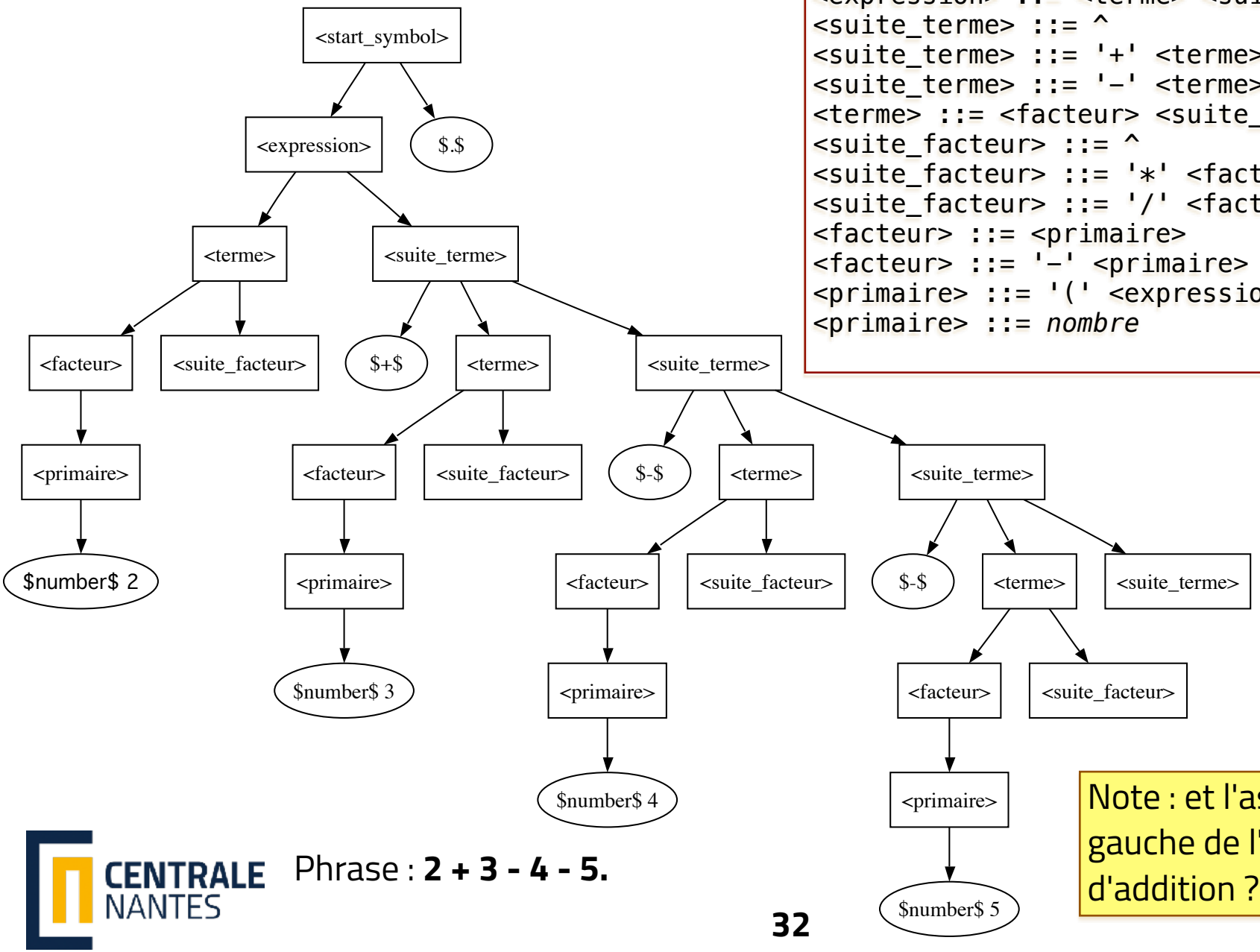
Arbre Syntaxique Concret

Un analyseur syntaxique construit l'arbre syntaxique concret de la phrase analysée.

Si la grammaire est non ambigüe, l'arbre est unique pour une phrase donnée (et ne dépend pas de la classe de la grammaire).

Exemple d'arbre syntaxique concret

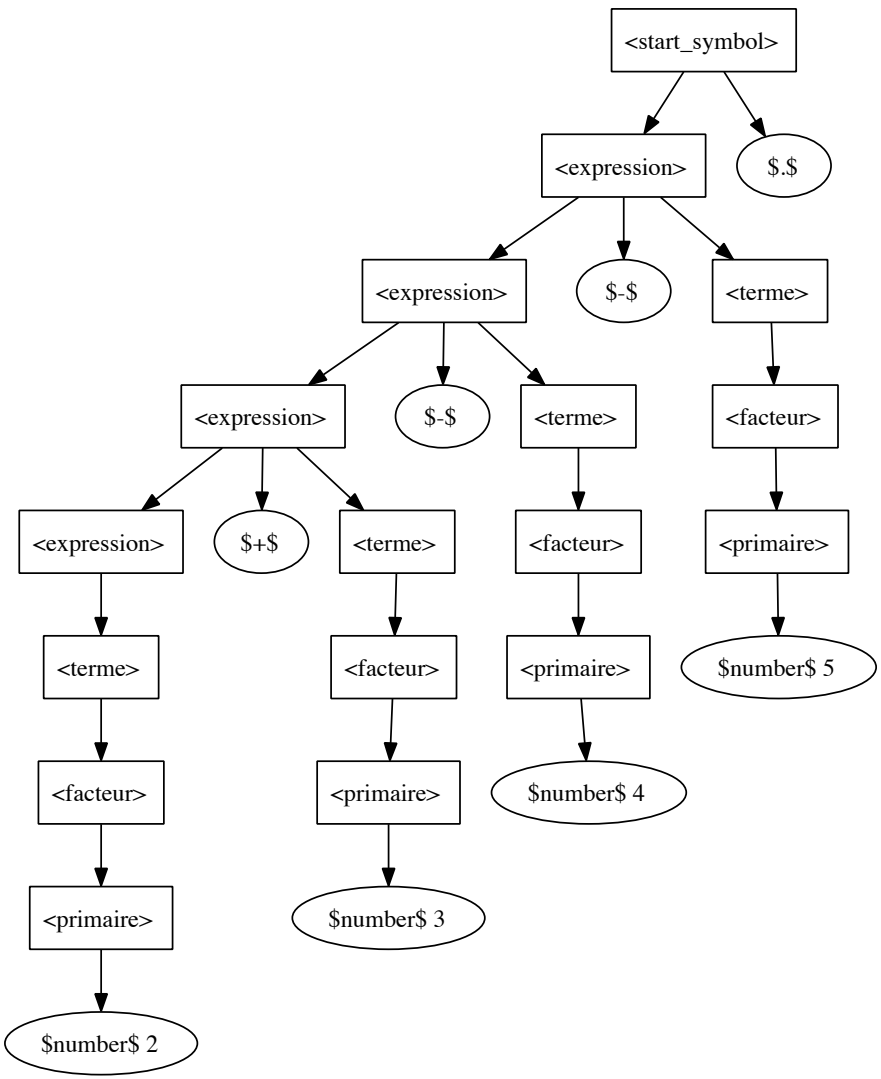
```
<axiome> ::= <expression> '.'
<expression> ::= <terme> <suite_terme>
<suite_terme> ::= ^
<suite_terme> ::= '+' <terme> <suite_terme>
<suite_terme> ::= '-' <terme> <suite_terme>
<terme> ::= <facteur> <suite_facteur>
<suite_facteur> ::= ^
<suite_facteur> ::= '*' <facteur> <suite_facteur>
<suite_facteur> ::= '/' <facteur> <suite_facteur>
<facteur> ::= <primaire>
<facteur> ::= '-' <primaire>
<primaire> ::= '(' <expression> ')'
<primaire> ::= nombre
```



Phrase : **2 + 3 - 4 - 5.**

Note : et l'associativité à gauche de l'opérateur d'addition ?

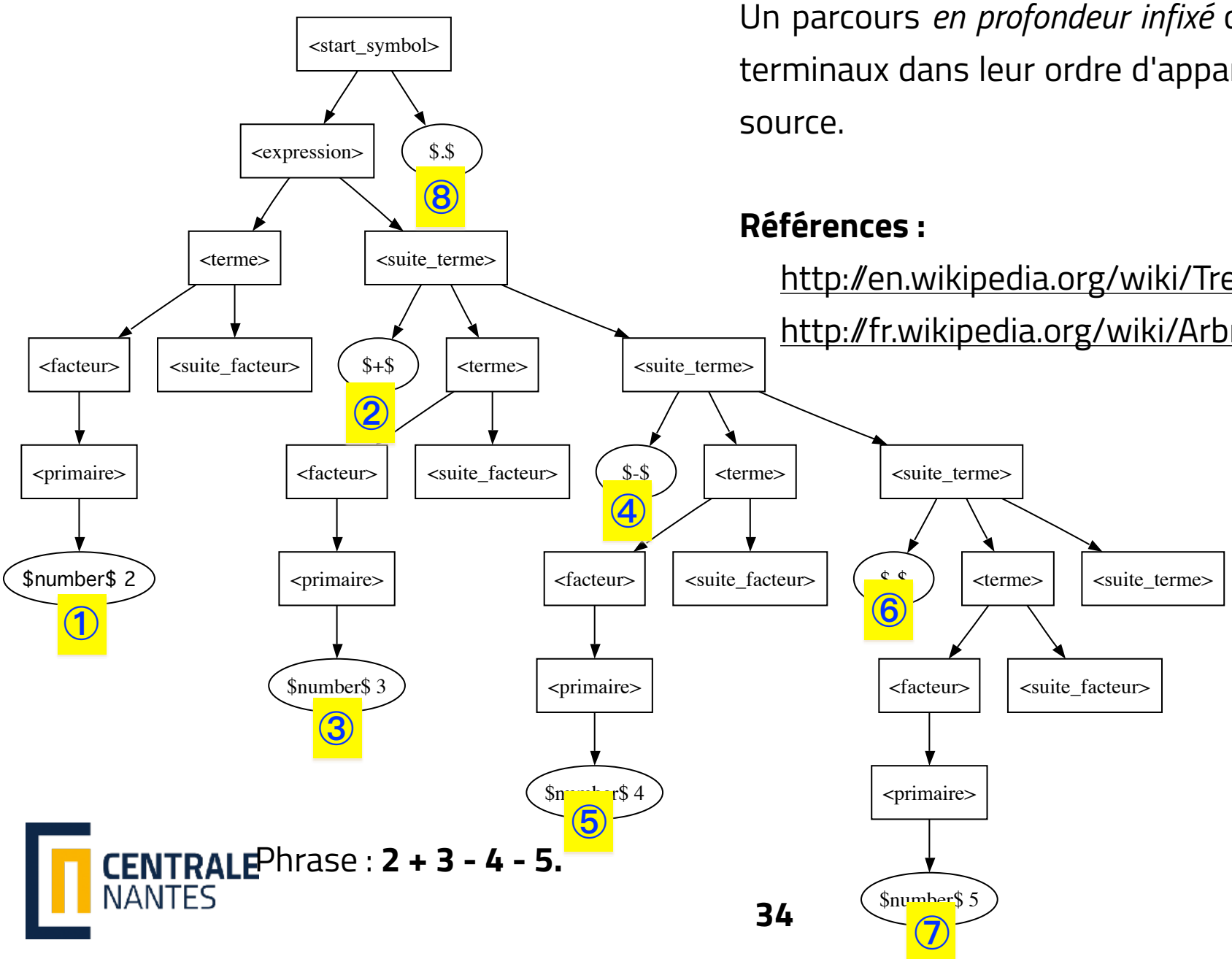
Autre exemple d'arbre syntaxique concret : même phrase, autre grammaire




```
<axiome> ::= <expression> '.'
<expression> ::= <expression> '+' <terme>
<expression> ::= <expression> '-' <terme>
<expression> ::= <terme>
<terme> ::= <terme> '*' <facteur>
<terme> ::= <terme> '/' <facteur>
<terme> ::= <facteur>
<facteur> ::= <primaire>
<facteur> ::= '-' <primaire>
<primaire> ::= '(' <expression> ')'
<primaire> ::= nombre
```

Phrase : **2 + 3 - 4 - 5.**

Parcours en profondeur infixé de l'arbre syntaxique concret



Un parcours *en profondeur infixé* de l'arbre visite les terminaux dans leur ordre d'apparition dans le texte source.

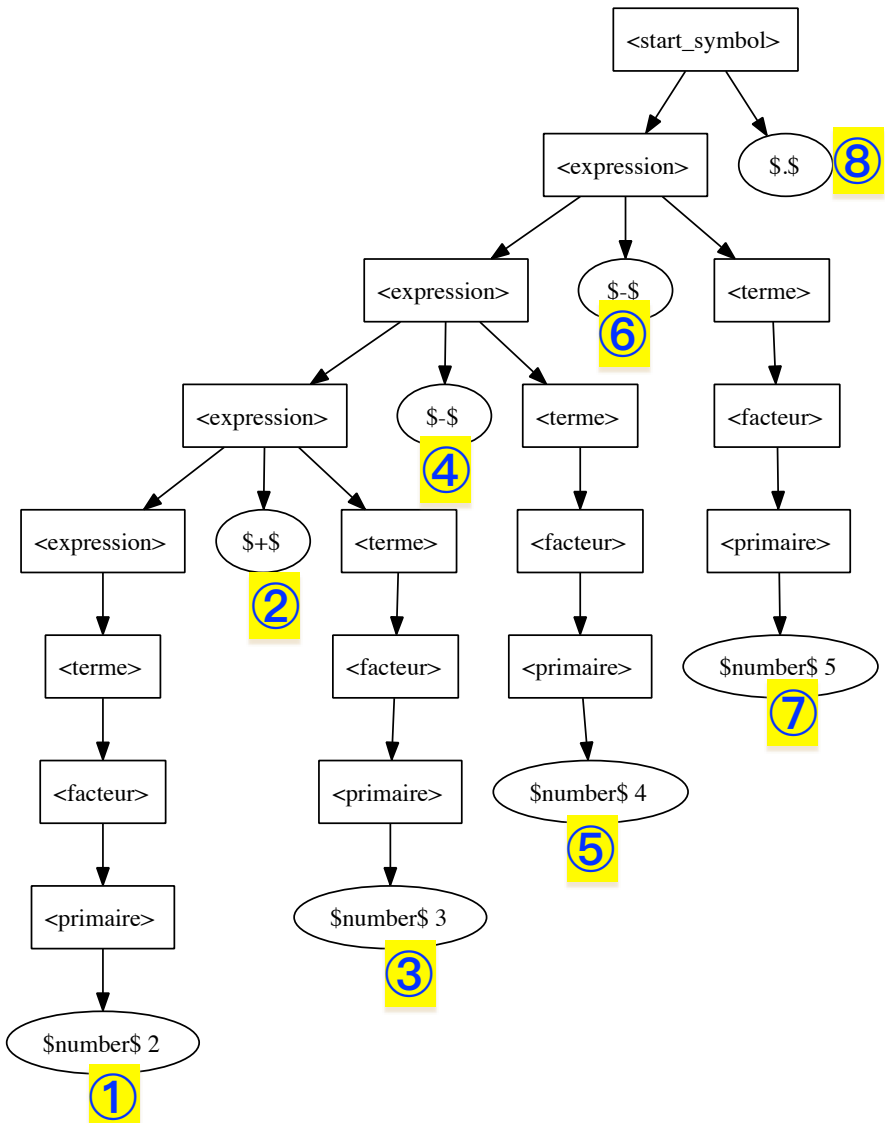
Références :

http://en.wikipedia.org/wiki/Tree_traversal

http://fr.wikipedia.org/wiki/Arbre_(informatique)

Phrase : **2 + 3 - 4 - 5.**

Parcours de l'arbre syntaxique concret, autre grammaire



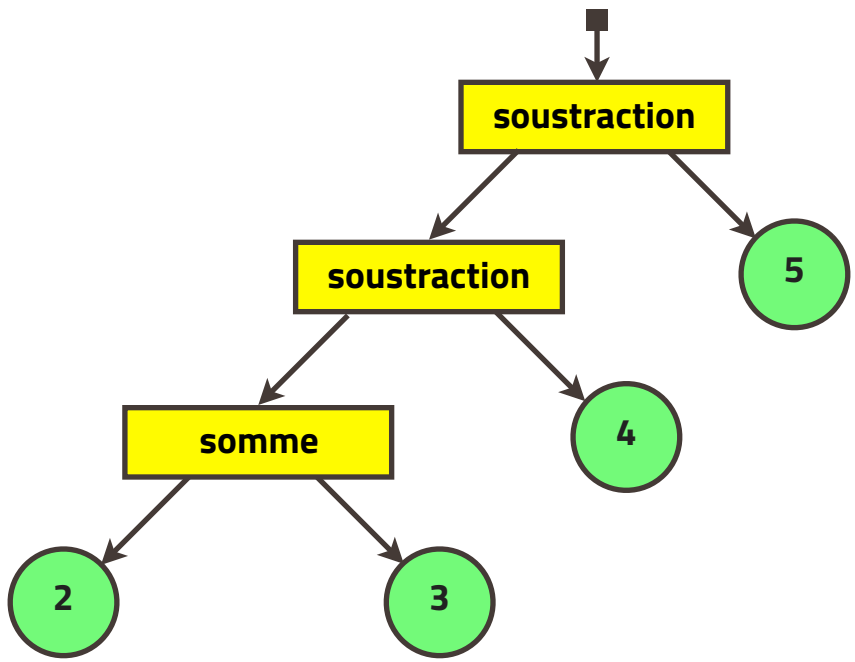
Phrase : **2 + 3 - 4 - 5.**

Arbre syntaxique concret, discussion

L'arbre syntaxique concret n'est pas très utile pour exprimer la sémantique :

- il comporte des détails sans importance sémantique ;
- il est trop lié aux règles de production de la grammaire ;
- les nœuds ne sont pas typés.

Ce que l'on voudrait obtenir pour la phrase « **2 + 3 - 4 - 5.** », et ceci pour les deux grammaires :



Un tel arbre est appelé *arbre syntaxique abstrait*.

Ce sont les grammaires attribuées qui indiquent comment construire un tel arbre.

Grammaires attribuées

Le concept de grammaire attribuée a été proposée par Knuth (1968) :

Semantics of Context-free Languages. Mathematical Systems Theory, 2(2):127-145. Springer-Verlag, 1968

Référence :

http://en.wikipedia.org/wiki/Attribute_grammar

C'est l'article fondateur du domaine. Que dit cet article ? Qu'un calcul peut être entrepris en parcourant l'arbre syntaxique concret. Ce calcul est spécifié en rajoutant des règles de calcul aux règles de production. Les valeurs calculées par ces règles sont nommées *attributs*, elles accompagnent le parcours de l'arbre :

- un attribut *hérité* descend l'arbre ;
- un attribut *synthétisé* remonte l'arbre.

Les différents types de grammaires attribuées

L-attributed-grammars : elles peuvent être évaluées par un parcours en profondeur de gauche à droite de l'arbre syntaxique, et sont donc adaptés aux grammaires analysées par descente récursive.

S-attributed-grammars : ces grammaires n'ont pas d'attributs hérités, que des attributs synthétisés, et sont donc adaptés aux grammaires analysées de manière ascendante.

LR-attributed-grammars : ces grammaires peuvent être évaluées au cours d'une analyse syntaxique LR (donc adaptées aux grammaires LR (1)).

Références :

http://en.wikipedia.org/wiki/L-attributed_grammar
http://en.wikipedia.org/wiki/S-attributed_grammar
http://en.wikipedia.org/wiki/LR-attributed_grammar

Utilisation des grammaires attribuées

Il y a deux principales utilisations des grammaires attribuées.

Construire l'arbre syntaxique abstrait. Uniquement des attributs synthétisés sont mis en œuvre, de façon à « remonter » et à « combiner » les arbres partiels obtenus aux niveaux inférieurs.

Effectuer directement l'analyse sémantique. Cela suppose que le calcul d'attributs permet d'effectuer complètement ce travail, ce qui est possible pour des langages simples, qui obéissent à la règle « *déclaration avant utilisation* ».

Souvent, les compilateurs industriels sont construits suivant ce schéma, au prix de la définition dans le langage de constructions particulières pour contourner l'exigence de la règle « *déclaration avant utilisation* ».

Arbre syntaxique abstrait

De quels types a-t'on besoin pour décrire un arbre sémantique abstrait ?

A priori, du seul type **classe**.

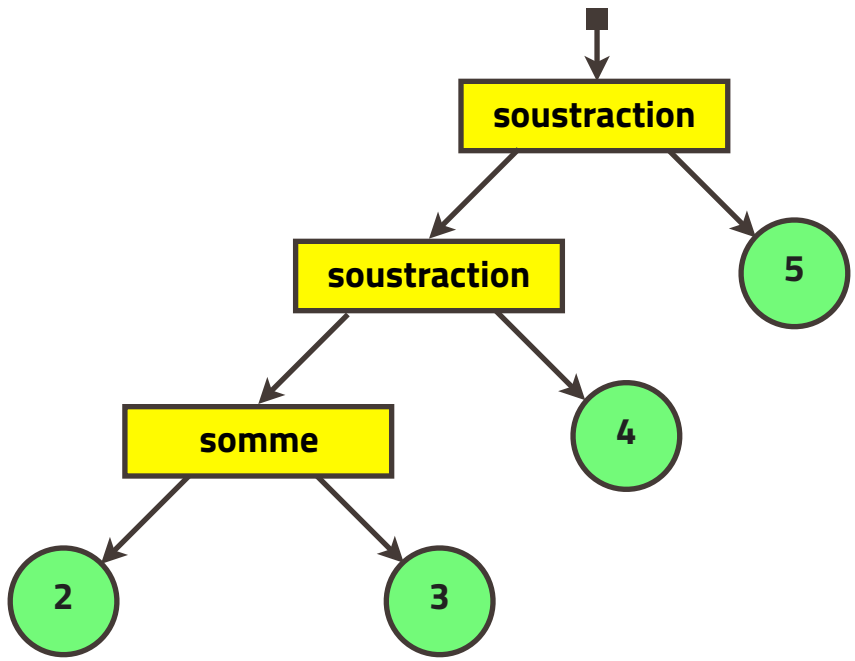
Exemple (les constructeurs et destructeurs ne sont pas représentés) :

```
class cExpression { … } ; // Classe abstraite

class cLiteralNumber : public cExpression {
  int mValue ;
  …
} ;

class cSum : public cExpression {
  cExpression * mLeftOperand ;
  cExpression * mRightOperand ;
  …
} ;
```

Construction d'un arbre syntaxique abstrait

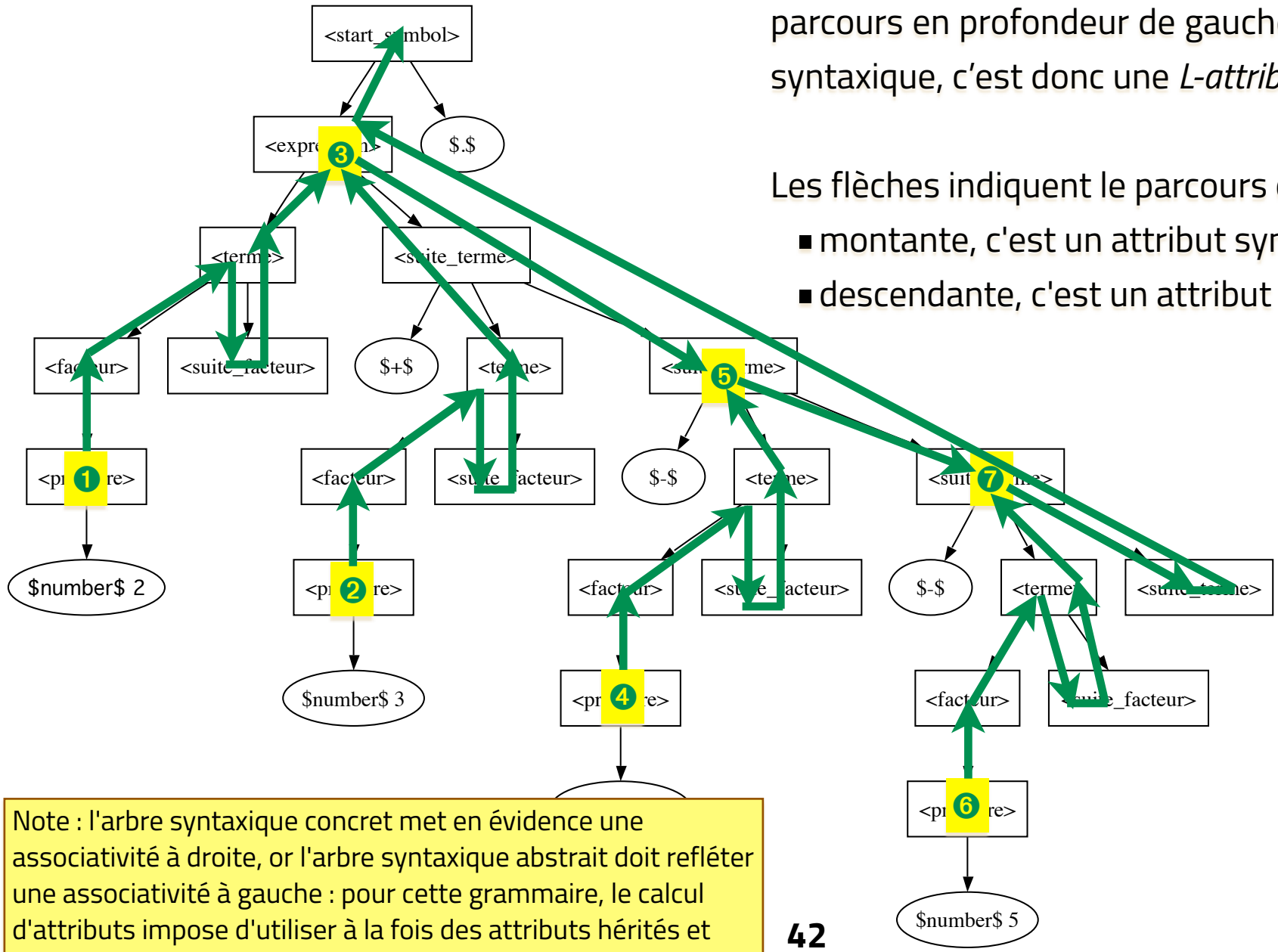
Exemple, l'arbre syntaxique abstrait correspondant à la phrase « **2 + 3 - 4 - 5.** » :

```
1 cExpression * op1 = new cLiteralNumber (2) ;
2 cExpression * op2 = new cLiteralNumber (3) ;
3 cExpression * op3 = new cSum (op1, op2) ;
4 cExpression * op4 = new cLiteralNumber (4) ;
5 cExpression * op5 = new cSoustraction (op3, op4) ;
6 cExpression * op6 = new cLiteralNumber (5) ;
7 cExpression * e = new cSoustraction (op5, op6) ;
```



CENTRALE NANTES

Comment est parcouru l'arbre syntaxique concret



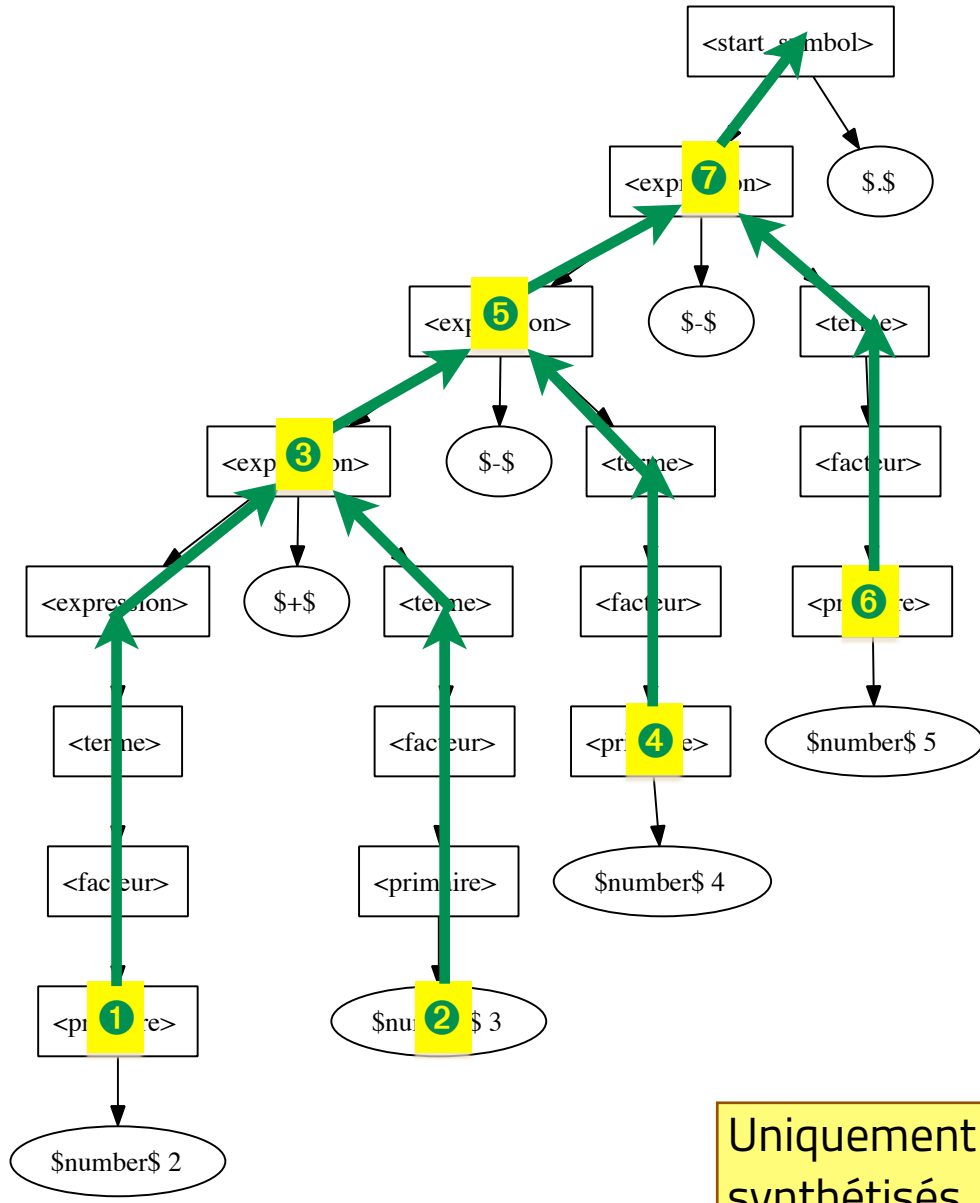
L'arbre syntaxique abstrait est évalué par un parcours en profondeur de gauche à droite de l'arbre syntaxique, c'est donc une *L-attributed-grammar*.

Les flèches indiquent le parcours des attributs :

- montante, c'est un attribut synthétisé ;
- descendante, c'est un attribut hérité.

Note : l'arbre syntaxique concret met en évidence une associativité à droite, or l'arbre syntaxique abstrait doit refléter une associativité à gauche : pour cette grammaire, le calcul d'attributs impose d'utiliser à la fois des attributs hérités et synthétisés.

Comment est parcouru l'arbre syntaxique concret, 2e exemple



Uniquement des attributs synthétisés.

Le type liste

Les classes suffisent pour construire l'arbre syntaxique abstrait.

D'autres types peuvent-ils être utiles ? Le type liste. En effet, la plupart des langages font apparaître des listes (d'instructions, de déclaration, ...)

Les classes permettent de représenter des listes, il suffit d'utiliser un attribut pour chaîner vers l'objet suivant de la liste.

Cependant, la représentation d'une liste via des classes :

- n'est pas naturelle ;
- il n'est pas possible d'utiliser des *itérateurs* sur des listes ainsi formées.

Opérations d'un type liste :

- instanciation d'une liste vide ;
- ajout d'un élément à la fin de la liste ;
- énumération de tous les éléments de la liste.

Les deux premières opérations sont invoquées pour la construction de l'arbre syntaxique abstrait, la dernière pour son parcours.

Astuce :
éliminer le sucre syntaxique

CENTRALE NANTES

Sucre syntaxique ?

Sucre syntaxique : constructions syntaxiques qui simplifient l'écriture des sources, mais qui ne sont pas indispensables.

Voici trois exemples :

- opérateurs C combinés avec une affectation ;
- l'opérateur « `!=` » ;
- les opérateurs C « && » et « || » ;
- instruction **`if … elsif … else … end`**.

opérateurs C combinés avec une affectation

Par exemple, en C, on peut écrire au choix :

- `a += b;`
- `a = a + b.`

Un *bon* compilateur doit engendrer le même code.

On confie donc à l'analyseur syntaxique le travail d'engendrer le même arbre syntaxique abstrait pour les deux constructions. Avantage : l'analyse sémantique est allégée.

Opérateur « != »

Sémantiquement : `a!=b` ⇔ `!(a==b)`

Donc, il suffit d'implémenter l'un des deux opérateurs.

Dans le code engendré, souvent implémenter la négation a un coût nul (voir la partie génération de code).

On confie donc à l'analyseur syntaxique le travail d'éliminer un des deux opérateurs au profit de l'autre. Avantage : l'analyse sémantique ne traite qu'un seul opérateur.

Opérateurs *court-circuit* C « && » et « || »

En C, `(a||b)` a pour sémantique :

- si `a` est vrai, `b` n'est pas évalué et l'expression est vraie ;
- si `a` est faux, `b` est évalué et la valeur de l'expression est la valeur de `b`.

Le C définit aussi l'opérateur &&, et `(a&&b)` a pour sémantique :

- si `a` est faux, `b` n'est pas évalué et l'expression est faux ;
- si `a` est vrai, `b` est évalué et la valeur de l'expression est la valeur de `b`.

Il ya redondance en ces deux opérateurs : `a||b` ⇔ `!(!a && !b)`

Dans le code engendré, souvent implémenter la négation a un coût nul (voir la partie génération de code).

On confie donc à l'analyseur syntaxique le travail d'éliminer un des deux opérateurs au profit de l'autre. Avantage : l'analyse sémantique ne traite qu'un seul opérateur.

Instruction `if … elsif … else … end`

Une instruction **`if`** typique accepte :

- zéro, une ou plusieurs branches **`elsif`** ;
- zéro ou une branche **`else`**.

On confie donc à l'analyseur syntaxique le travail de construire une instruction **`if`** sans branche **`elsif`** et toujours une branche **`else`** (qui peut contenir aucune instruction).

```
if a then
  A
elsif b then
  B
elsif c then
  C
else
  D
end
```

⟹

```
if a then
  A
else
  if b then
    B
  else
    if c then
      C
    else
      D
    end
  end
end
```

Analyse sémantique

Plusieurs parcours de l'arbre syntaxique abstrait

Il est classique d'effectuer plusieurs parcours de l'arbre syntaxique abstrait ordonné :

- le premier pour faire l'inventaire de toutes les entités déclarées ;
- le second pour effectuer l'analyse sémantique.

Exemple de fonctions récursives

```
function fA {
  …
  fB () ;
  …
}
```

```
function fB {
  …
  fA () ;
  …
}
```

Il n'y a pas de contraintes d'ordre entre **fA** et **fB**.

Note : si le langage n'accepte pas la récursivité, il suffit d'imposer une contrainte d'ordre entre une fonction et les fonctions qu'elle appelle.

Table de symboles

Une table de symboles est une structure de données associant une *information* à une *clé*.

Opérations :

- instanciation d'une table vide ;
- insertion d'une nouvelle clé associée à une information (avec déclenchement d'une erreur si la clé existe déjà) ;
- obtention de l'information associée à une clé (avec déclenchement d'une erreur si la clé n'existe pas) ; éventuellement, effectuer des suggestions en affichant les clés *voisines* au sens de la distance de Levenshtein.

Références :

http://en.wikipedia.org/wiki/Symbol_table

https://fr.wikipedia.org/wiki/Distance_de_Levenshtein

Implémentation des tables des symboles

Différentes techniques permettent d'implémenter les tables de symboles :

Références :

http://en.wikipedia.org/wiki/Binary_search_tree

http://en.wikipedia.org/wiki/AVL_tree

Implémentation	Insertion		Recherche	
	Moyenne	Pire cas	Moyenne	Pire cas
Tableau non ordonné	O (n)	O (n)	O (n)	O (n)
Tableau ordonné	O (n)	O (n)	O (log n)	O (log n)
Arbre binaire ordonné	O (log n)	O (n)	O (log n)	O (n)
Arbre binaire équilibré	O (log n)	O (log n)	O (log n)	O (log n)

Espaces de noms

Plusieurs choix :

- une table contenant tous les symboles ;
- plusieurs tables, chaque type de symbole ayant sa propre table.

Chaque table définit un espace de noms.

Par exemple :

une table pour les fonctions ;

une table pour les types ;

une table pour les variables ;

...

Une table contenant tous les symboles : c'est la solution la plus générale.

Une table pour chaque type de symbole : possibilité d'avoir des noms identiques dans plusieurs tables.

Exemple en C

En C, les structures peuvent avoir leur propre espace de noms, c'est-à-dire leur propre table.

```
typedef struct T {   ← Ce symbole appartient à la table des structures
  …
} U ;                ← Ce symbole appartient à la table générale des symboles
```

Quand il existe plusieurs tables de symboles, la syntaxe doit préciser dans quelle table le symbole doit être recherché. Ainsi :

```
U variable ;
```

U est recherché dans la table générale des symboles.

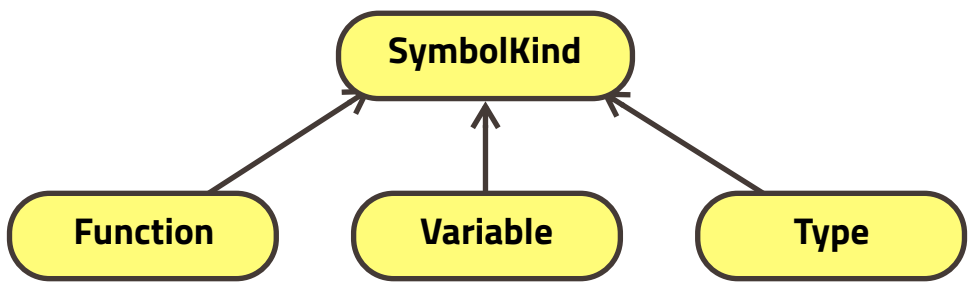
```
struct T variable ;
```

T est recherché dans la table des structures.

Table de symboles et espaces de noms

Imaginons un langage dans lequel variables, types et fonctions seraient dans des espaces différents : durant le contrôle sémantique, 3 tables de symboles seraient utilisées.

Par contre, si variables, types et fonctions sont dans le même espace de nommage, une seule table de symboles est utilisée. Une hiérarchie de classes permet d'établir la nature de chaque symbole.



L'information associée à chaque symbole est donc une instance de *SymbolKind*.

Vérifier la valuation des objets (1/4)

Quel est le bug du code suivant ?

```
int a ;
if (condition) {
  a = 5 ;
}
```

Un *bon* compilateur se doit de détecter ce type d'erreur. Par contre, pas d'erreur dans le code suivant :

```
int a ;
if (condition) {
  a = 5 ;
}else{
  a = 3 ;
}
```

Vérifier la valuation des objets (2/4)

Autre exemple :

```
int a = 8 ;
if (condition) {
  a = 5 ;
}else{
  a = 3 ;
}
```

La valeur 8 n'est pas utilisée, un compilateur devrait émettre un *warning*.

Vérifier la valuation des objets (3/4)

Une solution est d'associer lors de la compilation un automate à chaque variable.

C'est ce qui est fait en GALGAS, par exemple pour les variables locales.

Les états de l'automate :

- *Undefined local variable* : la variable est définie, mais n'a pas de valeur associée ;
- *Defined Local variable* : la variable est définie, a une valeur associée qui n'a pas été lue ;
- *Read local variable* : la variable est définie, a une valeur associée qui a été lue ;
- *Dropped local variable* : la variable est définie, a une valeur associée qui a été lue.

État initial :

- *Undefined local variable* : si la déclaration ne fournit pas de valeur initiale ;
- *Defined Local variable* : si la déclaration fournit une valeur initiale.

État final (examiné lorsque la variable locale est détruite) :

- cadre vert : pas d'erreur ;
- cadre orange : *warning* ;
- cadre rouge : erreur.

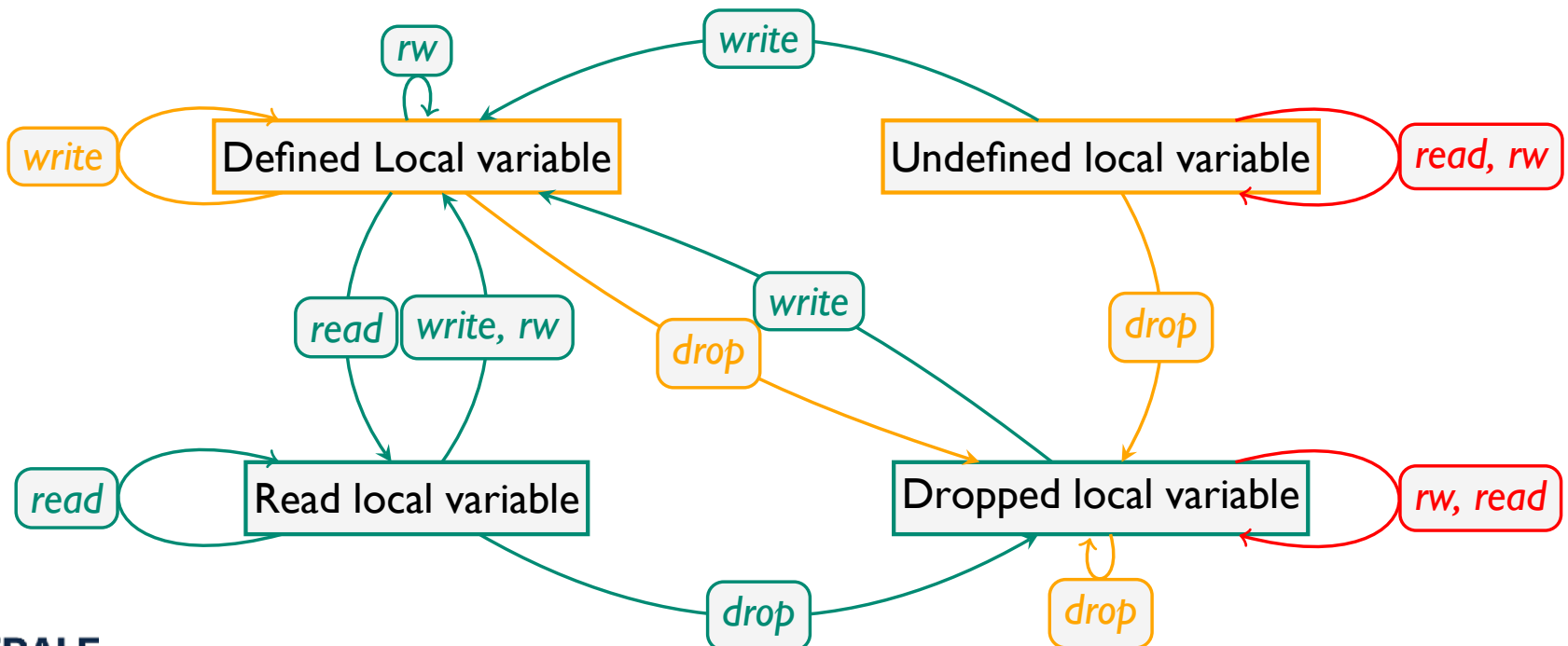
Vérifier la valuation des objets (4/4)

Les transitions de l'automate :

- *write* : écriture de la variable ;
- *read* : lecture de la variable ;
- *rw* : lecture suivie d'une écriture de la variable.

La couleur de la flèche indique la validité de l'accès :

- flèche verte : autorisé ;
- flèche orange : *warning* ;
- flèche rouge : erreur.



Flot de contrôle du code engendré

CENTRALE NANTES

Structuration par bloc

Pour obtenir un code optimisé, la meilleure solution est d'engendrer une séquence de **blocs**.

Un bloc est constitué :

- d'un point d'entrée ;
- d'une séquence (éventuellement vide) d'instructions ;
- d'une instruction de débranchement (saut, condition, ...).

L'intérêt de cette approche :

- facile à engendrer ;
- la sémantique ne dépend pas de l'ordre des blocs ;
- facile à optimiser.

La stratégie est la suivante :

(1) on engendre une liste de blocs ;

(2) élimination des blocs inutiles ;

(3) on réordonne les blocs pour éliminer les sauts au bloc suivant ;

(4) enfin, on engendre le code binaire.

Un exemple en Piccolo

Comment engendrer un code optimisé pour la routine ci-contre ?

```
noreturn routine main bank:requires 0 {
#--- Aucune entree analogique
  MOVLW 7
  MOVWF ADCON1
#--- Programmer RE0 en sortie
  BCF  TRISE.0
#--- Initialiser les compteurs
  CLRF compteurL
  CLRF compteurH
  CLRF compteurU
#--- Boucle infinie
  forever
    if (compteurL NZ)
      DECF compteurL
    elsif (compteurH NZ)
      DECF compteurH
      SETF compteurL
    elsif (compteurU NZ)
      DECF compteurU
      SETF compteurH
      SETF compteurL
    else
    #--- Réinitialiser les compteurs
      MOVLW  0x0F
      MOVWF  compteurU
      MOVLW  0x42
      MOVWF  compteurH
      MOVLW  0x40
      MOVWF  compteurL
    #--- Clignoter
      BTG  PORTE.0
    end
  end
}
```

Saut relatif inconditionnel en assembleur PIC18

BRA : saut relatif inconditionnel

Exemple de code engendré :

```
Addr.   Code    Assembly
00012   D00C    BRA .L4
…
0002C           .L4:
0002C   0601    DECF compteurH, F
```

BRA — **Unconditional Branch**

Syntax:	BRA n
Operands:	$-1024 \leq n \leq 1023$
Operation:	$(PC) + 2 + 2n \rightarrow PC$
Status Affected:	None
Encoding:	1101 | 0nnn | nnnn | nnnn
Description:	Add the 2's complement number, '2n', to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is a two-cycle instruction.
Words:	1
Cycles:	2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

Example: `HERE BRA Jump`

Before Instruction
PC = address (HERE)
After Instruction
PC = address (Jump)

Blocs engendrés pour `forever … end`

```
…
A
forever
   B
end
```

```
…
A
BRA .Lx
```

```
.Lx:
B
BRA .Lx
```

Saut conditionnel en assembleur PIC18

TSTFSZ : si le registre testé est nul, l'instruction suivante est ignorée.

Exemple de code engendré :

```
Addr.   Code    Assembly
00014   6602    TSTFSZ compteurU
00016   D00D    BRA .L7
00018   0E0F    MOVLW 0xF
…
00032           .L7:
```

TSTFSZ	**Test f, Skip if 0**
Syntax:	TSTFSZ f {,a}
Operands:	$0 \leq f \leq 255$ $a \in [0,1]$
Operation:	skip if f = 0
Status Affected:	None
Encoding:	0110 | 011a | ffff | ffff
Description:	If 'f' = 0, the next instruction fetched during the current instruction execution is discarded and a NOP is executed, making this a two-cycle instruction.

Cas général :

```
TSTFSZ reg
BRA .condition_fausse
BRA .condition_vraie
```

CENTRALE NANTES

Blocs engendrés pour `if … else … end`

Rappel : les branches **`elsif`** sont du sucre syntaxique, éliminées lors de l'analyse syntaxique.

```
…
A
if (reg NZ)
  B
else
  C
end
D
…
```

```
…
A
TSTFSZ reg
BRA .else
BRA .then
```

```
.then:
B
BRA .exit
```

```
.exit:
D
…
```

```
.else:
C
BRA .exit
```

Code intermédiaire engendré

```
LABEL .START, ORG 0x0:
  (JUMP main)

LABEL main:
  MOVLW 0x7 ; 7
  MOVWF ADCON1
  BCF TRISE, 0
  CLRF compteurL
  CLRF compteurH
  CLRF compteurU
  (JUMP .L0)

LABEL .L0:
  compteurL Z ? JUMP .L2 : JUMP .L1

LABEL .L1:
  DECF compteurL, F
  JUMP .L3

LABEL .L2:
  compteurH Z ? JUMP .L5 : JUMP .L4

LABEL .L4:
  DECF compteurH, F
  SETF compteurL
  JUMP .L6

LABEL .L5:
  compteurU Z ? JUMP .L8 : JUMP .L7

LABEL .L7:
  DECF compteurU, F
  SETF compteurH
  SETF compteurL
  JUMP .L9

LABEL .L8:
  MOVLW 0xF ; 15
  MOVWF compteurU
  MOVLW 0x42 ; 66
  MOVWF compteurH
  MOVLW 0x40 ; 64
  MOVWF compteurL
  BTG PORTE, 0
  (JUMP .L9)

LABEL .L9:
  (JUMP .L6)

LABEL .L6:
  (JUMP .L3)

LABEL .L3:
  JUMP .L0
```

Les parenthèses : saut vers le bloc qui suit, l'instruction de saut peut être omise.

CENTRALE NANTES

Suppression des blocs vides

Facile à optimiser : on supprime de manière itératives les blocs vides d'instructions.

① On court-circuite les blocs vides d'instructions.

```
LABEL .L1:
  DECF compteurL, F
  JUMP .L3
```



```
LABEL .L1:
  DECF compteurL, F
  JUMP .L0
```

```
LABEL .L3:
  JUMP .L0
```

② On élimine les blocs inaccessibles.

Résultat de la suppression :

```
Optimizations:
  Pass 1 : 14 blocks.
  Pass 2 : 12 blocks.
  Pass 3 : 11 blocks.
  Pass 4 : 9 blocks.
```

CENTRALE NANTES

Code intermédiaire après suppression des blocs vides

```
LABEL .START, ORG 0x0:
  (JUMP main)

LABEL main:
  MOVLW 0x7 ; 7
  MOVWF ADCON1
  BCF TRISE, 0
  CLRF compteurL
  CLRF compteurH
  CLRF compteurU
  (JUMP .L0)

LABEL .L0:
  compteurL Z ? JUMP .L2 : JUMP .L1

LABEL .L1:
  DECF compteurL, F
  JUMP .L0

LABEL .L2:
  compteurH Z ? JUMP .L5 : JUMP .L4

LABEL .L4:
  DECF compteurH, F
  SETF compteurL
  JUMP .L0

LABEL .L5:
  compteurU Z ? JUMP .L8 : JUMP .L7

LABEL .L7:
  DECF compteurU, F
  SETF compteurH
  SETF compteurL
  JUMP .L0

LABEL .L8:
  MOVLW 0xF ; 15
  MOVWF compteurU
  MOVLW 0x42 ; 66
  MOVWF compteurH
  MOVLW 0x40 ; 64
  MOVWF compteurL
  BTG PORTE, 0
  JUMP .L0
```

Ordonnancement des blocs

Choisir un *bon* ordre des blocs permet de supprimer des instructions de sauts.

Par exemple:

```
LABEL .L0:
  compteurL Z ? JUMP .L2 : JUMP .L1

LABEL .L1:
```

Traduction en assembleur

```
.L0:
  TSTFSZ compteurU
  BRA    .L1
  BRA    .L2
.L1 :
```

Un meilleur ordre, qui permet d'éliminer une instruction de saut :

```
LABEL .L0:
  compteurL Z ? (JUMP .L2) : JUMP .L1

LABEL .L2:
```

Traduction en assembleur

```
.L0:
  TSTFSZ compteurU
  BRA    .L1
.L2 :
```

Résultat :

```
Block ordering optimization:
  jump count before optimization: 7
  jump count after optimization: 4
```

CENTRALE NANTES

Code intermédiaire après ordonnancement des blocs

```
LABEL .START, ORG 0x0:
  (JUMP main)

LABEL main:
  MOVLW 0x7 ; 7
  MOVWF ADCON1
  BCF TRISE, 0
  CLRF compteurL
  CLRF compteurH
  CLRF compteurU
  (JUMP .L0)

LABEL .L0:
  compteurL Z ? (JUMP .L2) : JUMP .L1

LABEL .L2:
  compteurH Z ? (JUMP .L5) : JUMP .L4

LABEL .L5:
  compteurU Z ? (JUMP .L8) : JUMP .L7
```

```
LABEL .L8:
  MOVLW 0xF ; 15
  MOVWF compteurU
  MOVLW 0x42 ; 66
  MOVWF compteurH
  MOVLW 0x40 ; 64
  MOVWF compteurL
  BTG PORTE, 0
  JUMP .L0

LABEL .L1:
  DECF compteurL, F
  JUMP .L0

LABEL .L4:
  DECF compteurH, F
  SETF compteurL
  JUMP .L0

LABEL .L7:
  DECF compteurU, F
  SETF compteurH
  SETF compteurL
  JUMP .L0
```