Programmation Dirigée par la Syntaxe (PDS)

CM1 - Compilation & Syntaxe abstraite

ISTIC, Université de Rennes 1
Sebastien.Ferre@irisa.fr

PDS, M1 info

Remarques sur ce cours

- Évolution cours de compilation (COMP $\rightarrow$ PDS)
  - meilleure distinction (essentielle) entre syntaxe concrète et syntaxe abstraite
  - centrage sur la syntaxe abstraite
    - point de départ des phases essentielles de la compilation
    - *Compilation* $\subseteq$ *Programmation Dirigée par la Syntaxe (PDS)*
  - ouverture au-delà de la compilation
    bases de données, langue naturelle, ...
- Positionnement dans votre formation
  - est fondé (en partie) sur Theorie des langages formels (L2)
  - approfondit et élargit Compilation de L3
  - est suivi par Vérification en M1-S2
    - PDS : arbres syntaxiques $\rightarrow$ graphes sémantiques
    - Vérification : analyse de ces graphes

Plan

Plan

Pourquoi étudier la compilation ?

- Un compilateur utilise une grande quantité de techniques
  - de *beaux algorithmes*, utilisés dans de nombreux autres contextes
- Résultat d'une réflexion de 60 ans
  - dans le sens de toujours plus d'*automatisation*
  - un problème crucial et bien posé
    - *«traduire des programmes d'un langage dans un autre en préservant la sémantique et en étant le plus efficace possible»*
  - les compilateurs sont des programmes très complexes et en même temps très sûrs
  - rôle important de la *formalisation*

Pourquoi étudier la compilation ?

- Un compilateur utilise une grande quantité de techniques
  - de *beaux algorithmes*, utilisés dans de nombreux autres contextes
- Résultat d'une réflexion de 60 ans
  - dans le sens de toujours plus d'*automatisation*
  - un problème crucial et bien posé
    - *«traduire des programmes d'un langage dans un autre en préservant la sémantique et en étant le plus efficace possible»*
  - les compilateurs sont des programmes très complexes et en même temps très sûrs
  - rôle important de la *formalisation*

Pourquoi étudier la compilation ?

- Approcher l'intimité des langages de programmation
  - mieux les comprendre pour mieux programmer ?
  - percevoir les limites de ce qui est calculable
    - délimitation de classes de problèmes et leurs solutions
- La jonction entre le génie logiciel et le système
  - le système est responsable du chargement des programmes (ex., édition des liens)
  - le compilateur est responsable de la production de programmes chargeables (ex., tables de symboles)
- Ne concerne pas que les *grands* langages de programmation universels
  - nombreux *petits* langages de script ou d'échange de données (ex., XML. JSON)
  - langages de bases de données (ex., SQL)
  - les mêmes techniques sont à l'œuvre !

Pourquoi étudier la compilation ?

- Approcher l'intimité des langages de programmation
  - mieux les comprendre pour mieux programmer ?
  - percevoir les limites de ce qui est calculable
    - délimitation de classes de problèmes et leurs solutions
- La jonction entre le génie logiciel et le système
  - le système est responsable du chargement des programmes (ex., édition des liens)
  - le compilateur est responsable de la production de programmes chargeables (ex., tables de symboles)
- Ne concerne pas que les *grands* langages de programmation universels
  - nombreux *petits* langages de script ou d'échange de données (ex., XML. JSON)
  - langages de bases de données (ex., SQL)
  - les mêmes techniques sont à l'œuvre !

Pourquoi étudier la compilation ?

- Approcher l'intimité des langages de programmation
  - mieux les comprendre pour mieux programmer ?
  - percevoir les limites de ce qui est calculable
    - délimitation de classes de problèmes et leurs solutions
- La jonction entre le génie logiciel et le système
  - le système est responsable du chargement des programmes (ex., édition des liens)
  - le compilateur est responsable de la production de programmes chargeables (ex., tables de symboles)
- Ne concerne pas que les *grands* langages de programmation universels
  - nombreux *petits* langages de script ou d'échange de données (ex., XML. JSON)
  - langages de bases de données (ex., SQL)
  - les mêmes techniques sont à l'œuvre !

Ce que fait un compilateur

1. Analyse du programme source
   - trouver la structure du programme
   - signaler les éventuelles erreurs de syntaxe
   - formalisation par des règles de grammaire
   - structure = arbre de syntaxe abstraite (AST)
   - dépasse largement le cadre de la compilation
     - concerne toute application qui lit des entrées formatées selon une grammaire
2. Production de sa traduction
   - dans un langage cible
   - avec préservation de la sémantique (impératif)
   - et efficacité du code produit (autant que possible)

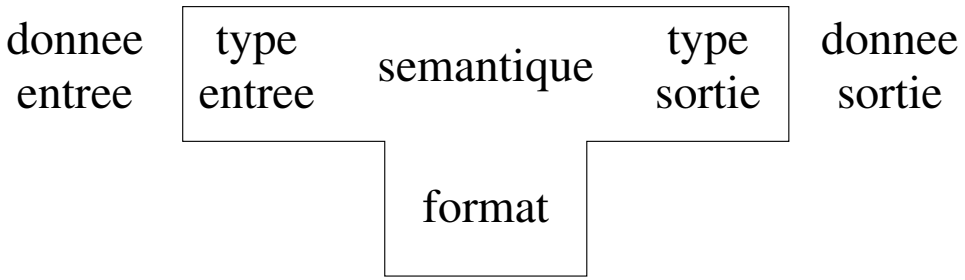
Donnée = Programme

Les données sont des programmes, et réciproquement

- point particulier des compilateurs
  - aussi à l'oeuvre dans les systèmes d'exploitation (gestion des processus)
- distinguer le programme «textuel» et le programme «en actes» (fonction)
- sémantique = lien entre le texte et la fonction

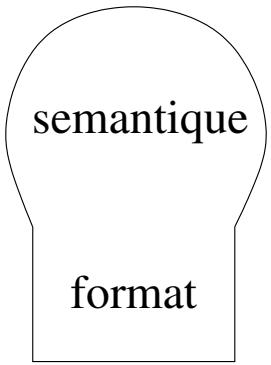
Diagramme en T

Représentation graphique d'un programme avec ses deux facettes : en actes/textuel



programme en actes

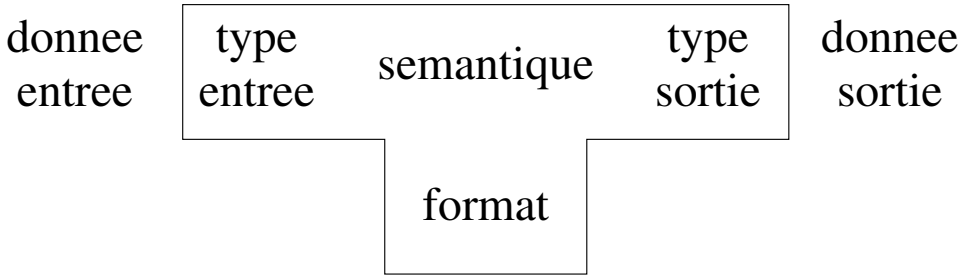
programme textuel



Ici, on ne detaille pas les types des entrees/sorties

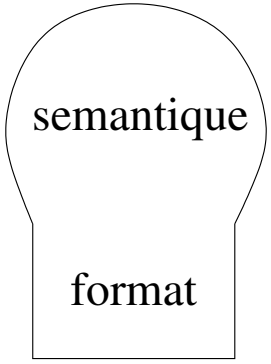
Diagramme en T

Représentation graphique d'un programme avec ses deux facettes : en actes/textuel



programme en actes

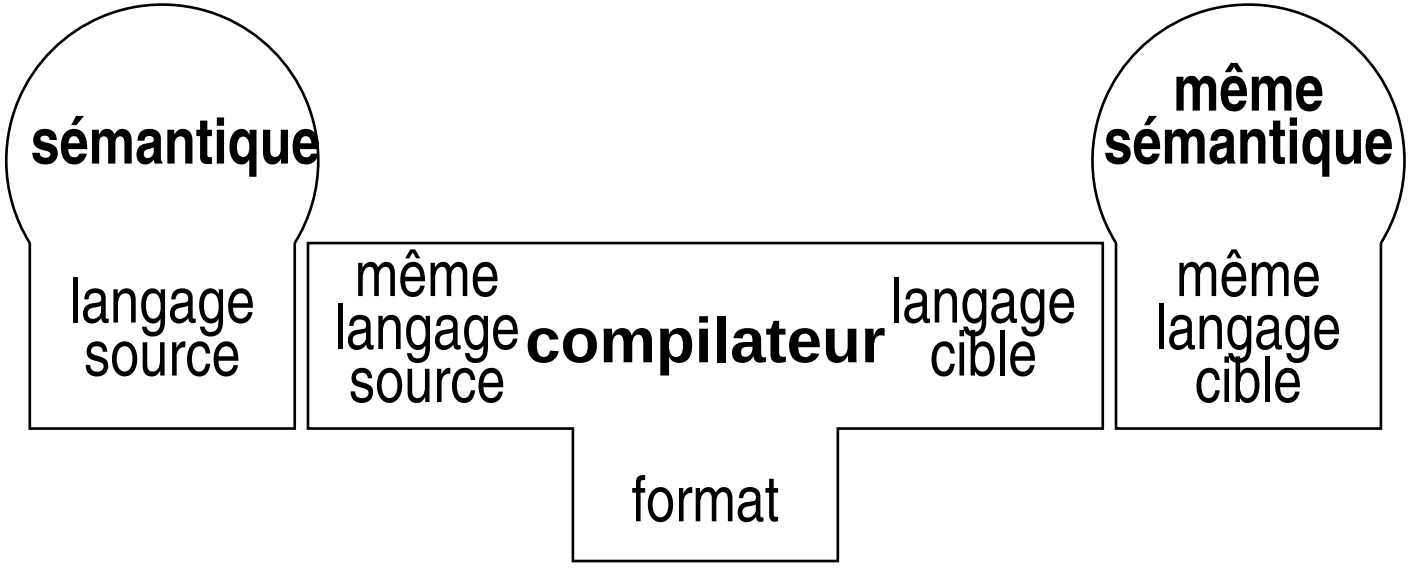
programme textuel



Ici, on ne detaille pas les types des entrees/sorties

Diagramme en T d'un compilateur

Relation entre compilateur et langages source et cible

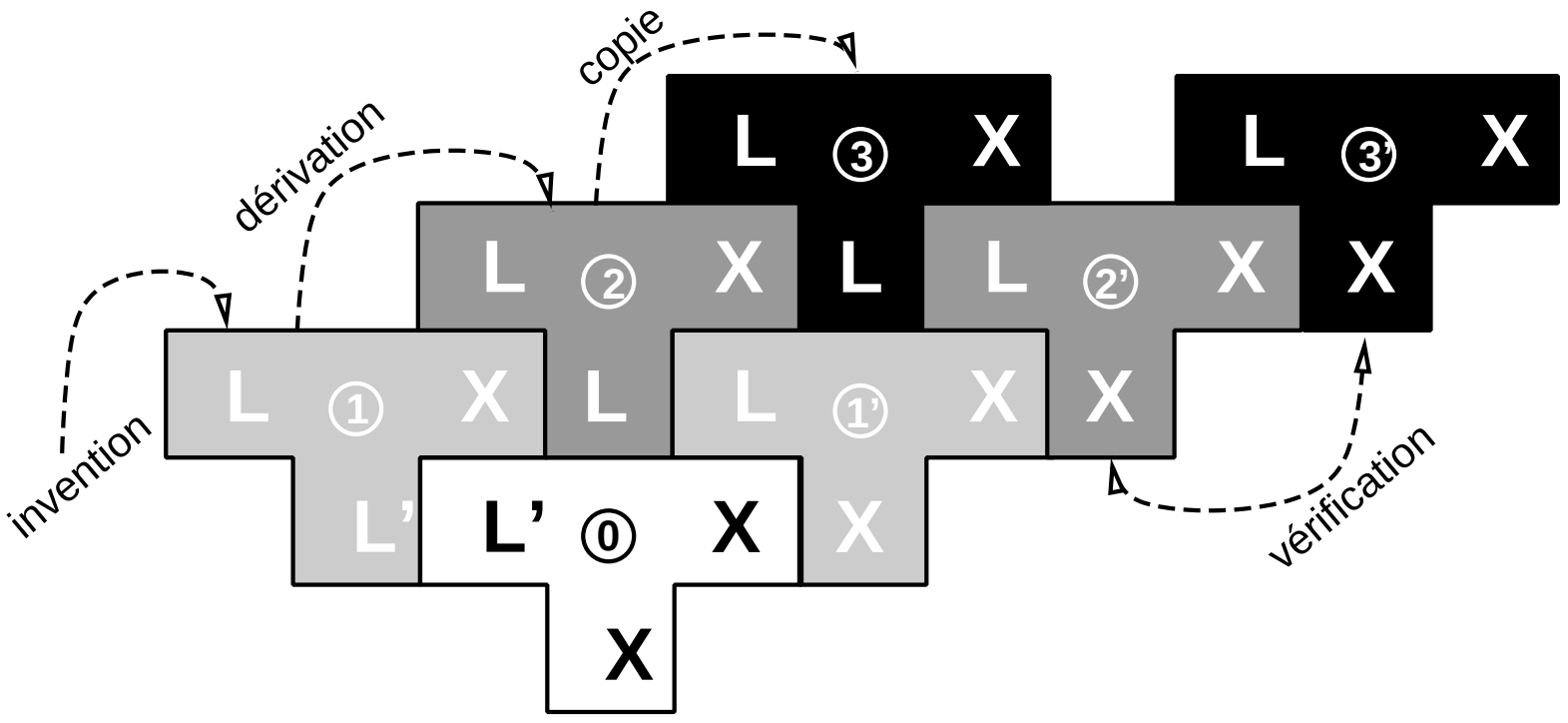


Note : les données d'entrée/sortie sont des programmes !

Auto-compilation (*bootstraping*)

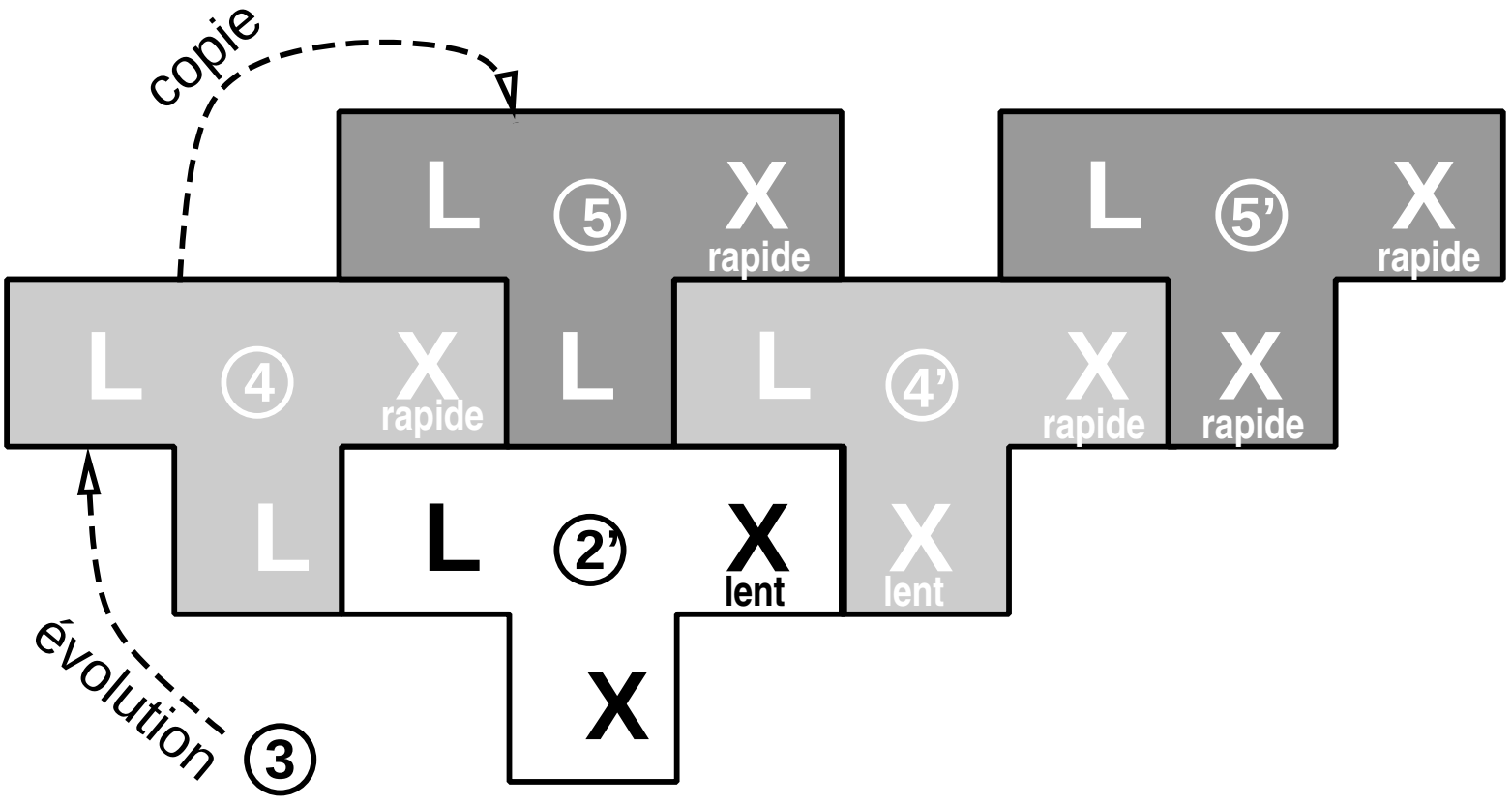
On souhaite écrire le compilateur dans son langage source.

- ex., écrire un compilateur Java en Java !
  L = Java, L' = C, X = langage machine éxécutable



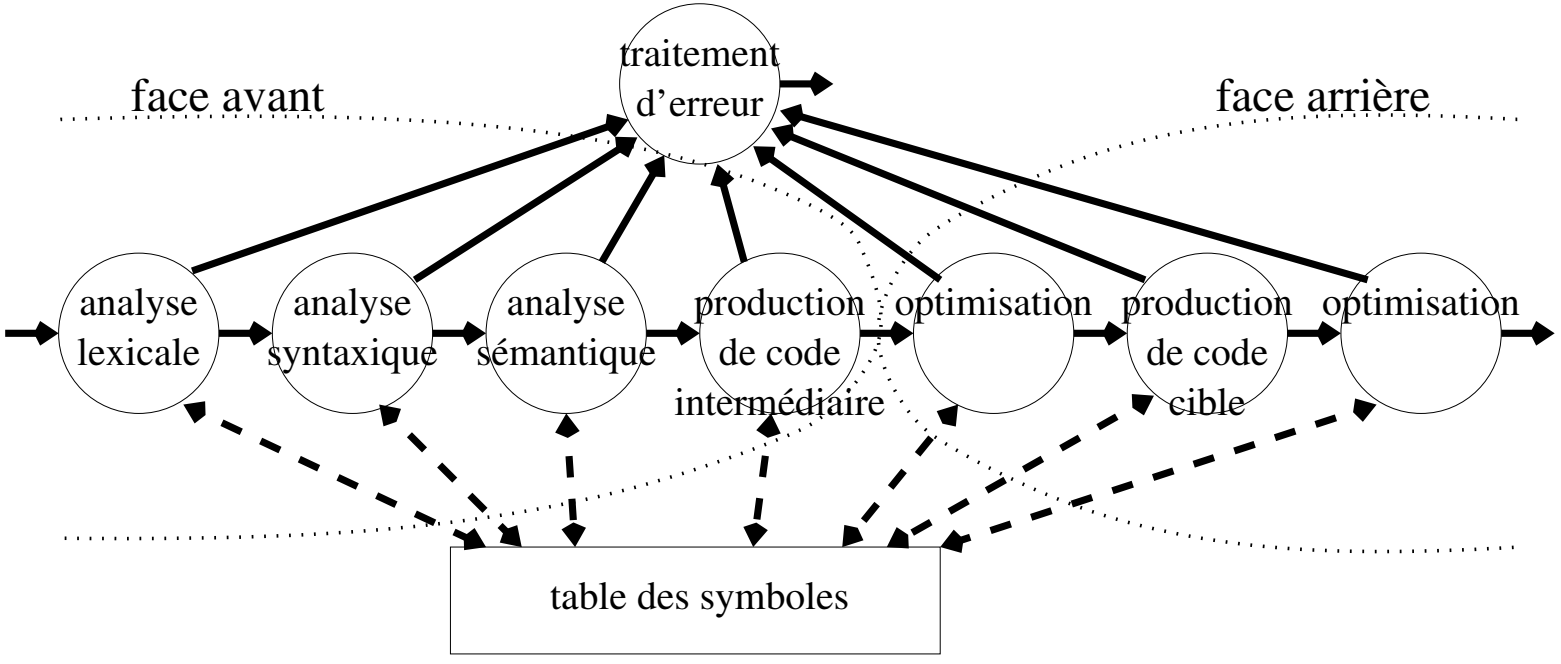
Évolution dans un schéma d'auto-compilation

On souhaite améliorer le compilateur (code plus rapide).



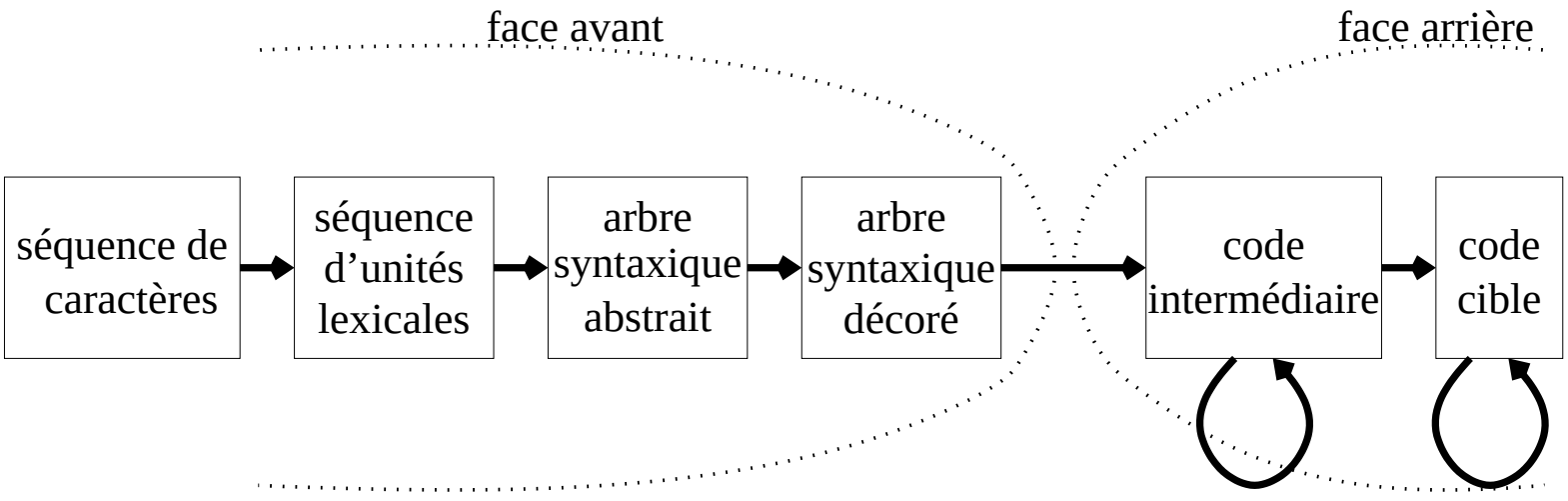
Les phases de la compilation

- elles forment un pipeline
- pas nécessairement en séquence stricte



Les représentations du programme

La représentation du programme évolue d'une phase à l'autre.



Analyses lexicales et syntaxiques

- Analyse lexicale
  - des lettres aux mots
  - langages formels de type 3
  - expressions régulières et automates finis
- Analyse syntaxique
  - des mots aux phrases (structure du programme)
  - langages formels de type 2
  - grammaires algébriques et automates à pile
  - + vérifications contextuelles (hors grammaire)
    - ex. : variables déclarées avant d'être utilisées

Analyses sémantiques

- détection des erreurs non-syntaxiques
- analogie : vérification des accords en nombre et en genre en français
- essentiellement : analyse de types
  - vérification vs inférence
  - typage *fort* vs *faible*
  - défini formellement dans les langages de type ML (ex., Objective Caml)

Production de code intermédiaire

- entre face avant et face arrière
- format relativement indépendant des langages source *et* cible
  - réutilisation de la face avant pour un autre langage cible (ex., une autre architecture)
  - réutilisation de la face arrière pour un autre langage source
  - factorisation des optimisations indépendantes de la cible

Optimisations et production de code

- la production du code intermédiaire est locale
  - traduction de chaque trait du langage source
  - compositionalité
  - code inefficace
- les optimisations ont souvent besoin d'une vision globale
  - ex. : savoir si une variable est utilisée après tel point de programme
  - techniques : analyse de flots de données, interprétation abstraite
- production du code cible
  - très lié à l'architecture (code binaire)
  - et au système d'exploitation (format des exécutables)

Structure du cours

- 8 CMs et 8 TDs
  1. Syntaxe abstraite
     représenter l'essence d'un langage
  2. Programmation dirigée par la syntaxe
     calculer sur des programmes
  3. Syntaxe concrète : pretty-printing et parsing
  4. Vérification de types
  5. Génération de code (expressions et instructions)
  6. Génération de code (structures de données et adressage)
  7. Graphes de flot de contrôle, de flot de données
     motivation : optimisations
  8. Application à l'interrogation de bases de données
     langages de requêtes

Structure du cours

- 8 TPs
  - TP1 (2 séances) : conversion de RDF/Turtle en RDF/Ntriples (1/3 note CC)
  - TP2 (6 séances) : réalisation d'un compilateur de VSL+ (2/3 note CC)
    - face avant : vérification de type et génération de code 3 adresses
    - face arrière fournie

Environnement pour les TPs

- Face avant : 2 propositions, au choix
  1. Java + ANTLR
     - ANTLR = ANother Tool for Language Recognition
  2. OCaml + stream parsers
     - suppose une connaissance de OCaml

  À choisir au TP1.
- Face arrière : LLVM = Low-Level Virtual Machine
  - production du code intermédiaire sous forme de texte par votre face avant
  - production d'un exécutable à partir du code intermédiaire par LLVM (fourni)

Plan

Syntaxe abstraite

Pourquoi ?

- représenter la structure syntaxique des phrases d'un langage
- faciliter la définition et l'éxécution de calculs sur ces phrases

Propriétés importantes

- précision
  - représenter toute l'information utile
  - ne laisser aucune ambiguité sur le sens des phrases
  - (1)……
- abstraction
  - ne représenter que l'information utile
  - éliminer les variations superficielles
  - (2)……
- simplicité
  - refléter les régularités du langage
  - pour éviter les duplications de code
  - pour limiter les risques de bugs
  - (3)……

Types pour la syntaxe abstraite

- langage = ens. de phrases
- syntaxe abstraite = ens. de structures syntaxiques (abstraites)
- parallèle : type = ens. de valeurs
  - ces valeurs peuvent être des structures de données, pas seulement des nombres ou des chaines !
  - les types définissent un cadre pour les calculs sur ces valeurs
    - entiers $\Rightarrow$ opérations arithmétiques
    - listes $\Rightarrow$ itérations, ajout/suppression

Types algébriques de données

Une syntaxe abstraite est définie par un type algébrique de données.

Definition

Un type algébrique de données (TAD) est un type complexe composé de types de bases :

- un TAD est défini par une union de variants
- un variant est
  - identifié par un constructeur
  - paramétré par des arguments
- un argument est
  - soit un type de base (ex., int, string)
  - soit un type complexe (le même TAD ou un autre)

Souvent, plusieurs TAD sont définis ensemble de façon mutuellement récursive

Exemple de TAD : langage d'expressions

(4)……

Valeurs d'un TAD

Un TAD est un type, donc il dénote un ensemble de valeurs.
Une valeur d'un TAD

- est une structure de donnée
- résultant de la composition
  - de constructeurs
  - et de valeurs de base (ex., entiers, chaines)
- selon la définition du TAD
  - chaque argument de constructeur doit être une valeur du type de l'argument dans le TAD

Exemples de (non-)valeurs de TAD : expressions

(5). . . . . .

ASD = Abstract Syntax Definition

Definition

Une Définition de Syntaxe Abstraite (Abstract Syntax Definition, ASD) est un ensemble de définitions de TAD couvrant l'ensemble des structures syntaxiques du langage visé.

Non-ambiguité

Une bonne ASD d'un langage *L* est telle que

1. toute phrase de *L* correspond à une et une seule valeur de l'ASD
   sa structure syntaxique
   - si plusieurs, l'ASD est ambigü, il manque de précision
   - si aucune, l'ASD est incomplet
2. toute valeur de l'ASD correspond à une ou plusieurs phrases de *L*
   ses formes concrètes
   - si aucune, l'ASD est non-strict
     - pas tragique en soi
     - mais va amener à définir des calculs (ex., génération de code) pour des cas qui n'existent pas dans le langage

Exemples d'ASD non-strictes

(6)……

Arbres de syntaxe abstraite (AST)

Definition

Un arbre de syntaxe abstraite (Abstract Syntax Tree, AST) est la représentation arborescente d'une valeur d'ASD.

- noeud = constructeur
- feuille = valeur d'un type de base
- fils d'un noeud = argument d'un constructeur

Chaque noeud peut être décoré par son type

- déductible du constructeur

Intérêt :

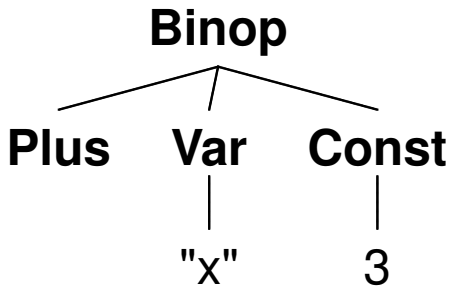
- mieux exposer la structure composite des syntaxes abstraites

Exemple d'AST

(7)……

Remarques

- ne pas confondre ASD et AST
  - c'est comme confondre "entier" et "42" !
  - ASD : description d'un langage
  - AST : description d'une phrase d'un langage
  - ASD/AST = type/valeur
  - l'AST d'une phrase d'un langage est une valeur de l'ASD du langage
- ne pas confondre l'AST et ses notations
  - 1 structure de données et plusieurs notations
    - textuelle/linéaire : **Binop**(**Plus**, **Var**("x"), **Const**(3))
    - graphique/arbre :



Syntaxe abstraite et syntaxe concrète

Différences :

- fonction :
  - **abstraite** : faire des calculs dirigés par la syntaxe
  - **concrète** : donner une forme de surface au langage
- implémentation :
  - **abstraite** : structures de données
  - **concrète** : séquences, analyseurs syntaxiques

Similarités

- forme :
  - **abstraite** : types algébriques de données
  - **concrète** : grammaires algébriques (hors-contexte)
    voir théorie des langages

Parallèle entre ASD et grammaire

ASD	grammaire
TAD	règle de grammaire
type	symbole non-terminal
type de base	unité lexicale, symbole terminal
variant	production, séquence de symboles
constructeur	-

Exemple de grammaire : expressions

(8). . . . . .

Autres différences

La grammaire, comparé à l'ASD

- est plus complexe
  - désambiguation, priorité et associativité des opérateurs
    plusieurs non-terminaux pour les expressions
  - parenthèses
- contient des détails supplémentaires
  - position des opérateurs : infixe, préfixe, postfix, ...
    opérateurs (+, -, *) infixes
  - ponctuations

Implémentation des syntaxes abstraites

- choix d'un langage de programmation
  - diagramme en T : langage du compilateur
- ASD $\Rightarrow$ ensemble de définitions de types
  - diagramme en T : langage source
- AST $\Rightarrow$ représentation en mémoire des valeurs des types définis
  - diagramme en T : donnée d'entrée
- style d'implémentation
  - variable en fonction du langage cible
  - mais très régulier (en fait, automatisable !)

(9)……

Langages d'implémentation

On considère ici deux familles modernes de langages de programmation :

1. famille ML : OCaml
2. famille OO : Java

Implémentation en ML/OCaml

- Le système de types des langages ML est basé sur les TAD (types algébriques de données)
- Le codage d'une ASD est donc direct, à quelques différences de notation près

Exemple : ASD des expressions en OCaml

(10). . . . . .

Exemple : AST d'une expression en OCaml

(11). . . . . .

Implémentation en OO/Java

- Les TAD ne sont pas natifs en programmation OO
- Mais ils sont facilement encodés avec le design pattern Composite
  - type $\Rightarrow$ classe abstraite
  - constructeur $\Rightarrow$ classe concrète
  - argument $\Rightarrow$ attribut de classe

Exemple : ASD des expressions en Java (1/2)

(12). . . . . .

Exemple : ASD des expressions en Java (2/2)

(12’).....

Exemple : AST d'une expression en Java

(13)……

Types énumérés

Un type d'une ASD dont tous les variants ont zéro argument peut être encodé par un type énuméré plutôt que par des classes

- avantage : code plus concis, représentation mémoire plus compacte
- inconvénient : moins extensible (ex. ajout d'arguments, méthodes)

Arguments optionnels et multiples

- Il est fréquent que certains arguments d'un variant soient
  - optionnels : 0 ou 1 valeur
  - multiples : 0 à *n* valeurs
- Exemples :
  - optionnel : branche "else" d'une conditionnelle
  - multiple : instructions dans un bloc

Exemple : instructions d'un langage impératif

$$
\begin{array}{lcl}
\textit{stat} & ::= & \textbf{Assign}(\textit{var}, \textit{expr}) \\
& | & \textbf{Conditional}(\textit{expr}, \textit{stat}, \textit{statOption}) \\
& | & \textbf{Block}(\textit{statList}) \\
\textit{statOption} & ::= & \textbf{Some}(\textit{stat}) \\
& | & \textbf{None} \\
\textit{statList} & ::= & \textbf{Cons}(\textit{stat}, \textit{statList}) \\
& | & \textbf{Nil}
\end{array}
$$

Types algébriques étendus

- Le codage en TAD des arguments optionnels/multiples est fastidieux
  - duplication des types *xOption*/*xList* pour chaque type *x*
- Extension des TAD par deux opérateurs sur les types
  - $x?$ = *xOption*
    - notation valeur manquante : ϵ
  - $x*$ = *xList*
    - notation liste : $[v_1, \ldots, v_n]$, []

Retour sur l'exemple

(15). . . . . .

Implémentation en ML/OCaml

- optionnel
  - argument $x?$ $\Rightarrow$ x `option`
  - valeur présente v $\Rightarrow$ `Some` v
  - valeur manquante ϵ $\Rightarrow$ `None`
- multiple
  - argument $x*$ $\Rightarrow$ x `list`
  - liste vide [] $\Rightarrow$ `[]`
  - list $[v_1, \ldots, v_n]$ $\Rightarrow$ `[`v_1`; ...; `v_n`]`

Exemple en OCaml

(16). . . . . .

Implémentation en OO/Java

- optionnel
  - argument $x?$ $\Rightarrow$ x
  - valeur présente v $\Rightarrow$ v
  - valeur manquante ϵ $\Rightarrow$ `null`
- multiple
  - argument $x*$ $\Rightarrow$ `List<`x`>`
  - liste $[v_1, \ldots]$ $\Rightarrow$ `(new List()).add(`v_1`)...`

Exemple en Java (1/2)

```
abstract class Statement { }

class Assign extends Statement {
    String var;
    Expr expr;
    public Assign(v,e) {
        var = v;
        expr = e; } }
class Cond extends Statement {
    Expr cond;
    Statement s_then;
    Statement s_else;
    public Cond(c,t,e) {
        cond = c;
        s_then = t;
        s_else = e; }
    public Cond(c,t) { // missing else branch
        cond = c;
        s_then = t;
        // s_else = null; } }
```

Exemple en Java (2/2)

```
class Block extends Statement {
    List<Statement> stats;
    public Block(l) {
        stats = l; } }

// creating the AST
List<Statement> l = new LinkedList();
l.add(new Assign("x", new Const(1)));
Statement s = new Cond(new Comp(new Geq(),
                                new Var("x"),
                                new Const(0)),
                       new Block(l),
                       null);
```

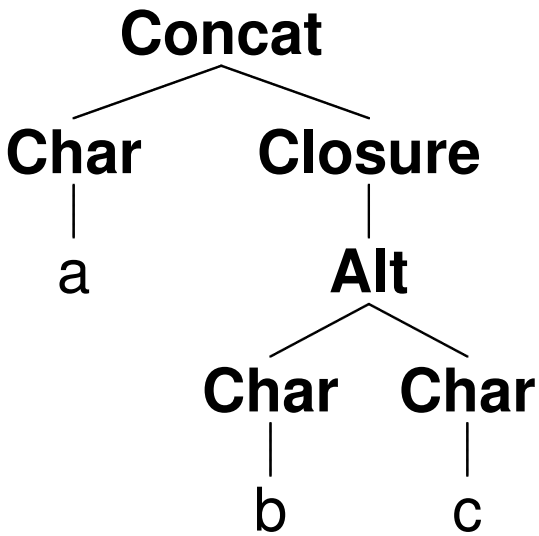
Plan

Expressions régulières : ASD

$$
\begin{array}{lll}
\textit{regexp} & ::= & \textbf{Char}(\textit{char}) \\
& | & \textbf{Epsilon} \\
& | & \textbf{Concat}(\textit{regexp}, \textit{regexp}) \\
& | & \textbf{Alt}(\textit{regexp}, \textit{regexp}) \\
& | & \textbf{Closure}(\textit{regexp})
\end{array}
$$

Expressions régulières : AST

$a(b|c)*$

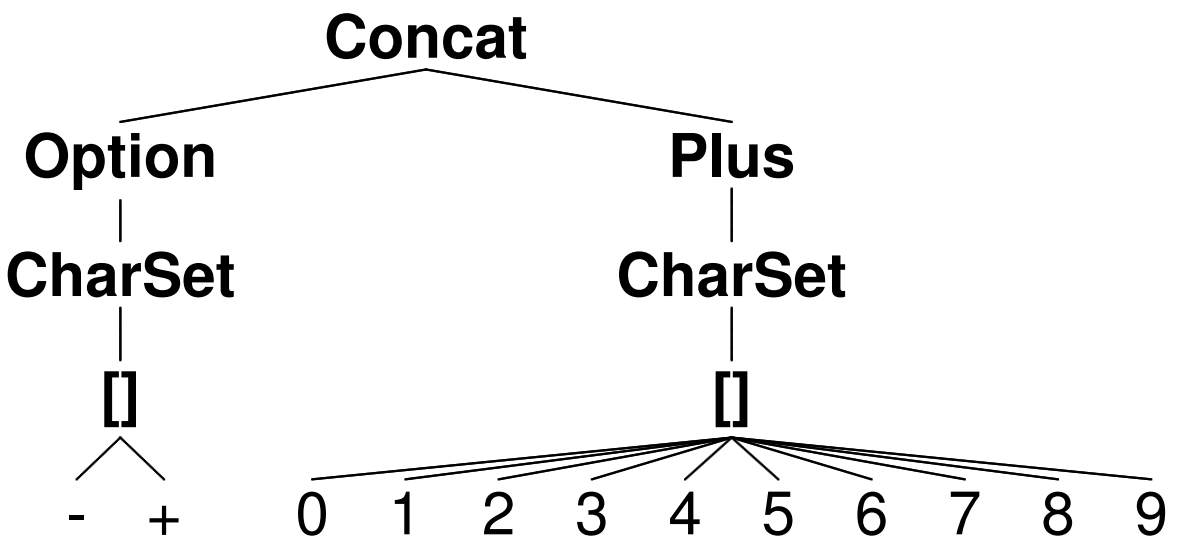


Expressions régulières étendues : ASD

$$
\begin{array}{lll}
\textit{regexp} & ::= & \textbf{Char}(\textit{char}) \\
& | & \textbf{Epsilon} \\
& | & \textbf{Concat}(\textit{regexp}, \textit{regexp}) \\
& | & \textbf{Alt}(\textit{regexp}, \textit{regexp}) \\
& | & \textbf{Closure}(\textit{regexp}) \\
& & \\
& | & \textbf{CharSet}(\textit{char}*) \\
& | & \textbf{Option}(\textit{regexp}) \\
& | & \textbf{Plus}(\textit{regexp}) \\
& | & \textbf{Interval}(\textit{regexp}, \textit{int}, \textit{int})
\end{array}
$$

Expressions régulières étendues : AST

```
[-+]?[0-9]+
```



Langage impératif : ASD 1/2

program	::=	**Prog**(*function*$*$)
function	::=	**Func**(*string*, *argument*$*$, *type*?, *statement*)
argument	::=	**Arg**(*string*, *type*)
type	::=	**Bool** | **Int** | **Float** | **String**
	|	**Pointer**(*type*)
statement	::=	**Assign**(*place*, *expression*)
	|	**SCall**(*string*, *expression*$*$)
	|	**Return**(*expression*?)
	|	**Block**(*declaration*$*$, *statement*$*$)
	|	**Cond**(*expression*, *statement*, *statement*?)
	|	**While**(*expression*, *statement*)
declaration	::=	**DeclVar**(*string*, *type*, *expression*?)

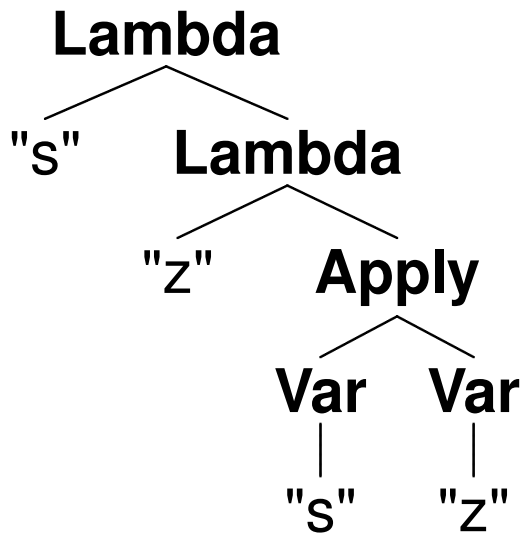
Langage impératif : ASD 2/2

place	::=	**Var**(*string*)
	|	**Deref**(*place*)
expression	::=	**BoolConst**(*bool*)
	|	**IntConst**(*int*)
	|	**FloatConst**(*float*)
	|	**StringConst**(*string*)
	|	**Null**
	|	**Value**(*place*)
	|	**Unop**(*unop*, *expression*)
	|	**Binop**(*binop*, *expression*, *expression*)
	|	**ECall**(*string*, *expression*$*$)
unop	::=	**UnaryMinus** | **Not**
binop	::=	**Plus** | **Minus** | **Times** | **Div** | **Mod**
	|	**Eq** | **Neq** | **Geq** | **Leq** | **Gt** | **Lt**
	|	**And** | **Or** | **Xor**

Lambda calcul : ASD + AST

$$
\begin{array}{lcl}
\textit{expression} & ::= & \textbf{Var}(\textit{string}) \\
& | & \textbf{Lambda}(\textit{string}, \textit{expression}) \\
& | & \textbf{Apply}(\textit{expression}, \textit{expression})
\end{array}
$$

$\lambda s.\lambda z.(s\ z)$



Langage fonctionnel : ASD

Sur-ensemble du lambda calcul

$$
\begin{array}{lcl}
\textit{program} & ::= & \textbf{Program}(\textit{definition}*) \\
\textit{definition} & ::= & \textbf{Define}(\textit{string}, \textit{expression}) \\
\textit{expression} & ::= & \textbf{BoolConst}(\textit{bool}) \\
 & | & \textbf{IntConst}(\textit{int}) \\
 & | & \textbf{FloatConst}(\textit{float}) \\
 & | & \textbf{StringConst}(\textit{string}) \\
 & | & \textbf{Pair}(\textit{expression}, \textit{expression}) \\
 & | & \textbf{Cons}(\textit{expression}, \textit{expression}) \mid \textbf{Nil} \\
 & | & \textbf{Var}(\textit{string}) \\
 & | & \textbf{Lambda}(\textit{string}, \textit{expression}) \\
 & | & \textbf{Apply}(\textit{expression}, \textit{expression})
\end{array}
$$

Soyons réflexifs !

Voici l'ASD des ASD

- *la syntaxe abstraite des syntaxes abstraites*

asd	::=	**ASD**(*typedef*$*$)
typedef	::=	**Typedef**(*string*, *variant*$*$)
variant	::=	**Variant**(*string*, *argument*$*$)
argument	::=	**One**(*string*)
	|	**Optional**(*string*)
	|	**List**(*string*)

Utilité de cet ASD :

- manipulation de syntaxes abstraites, donc de langages
- *compilateurs de compilateurs* (ex., ANTLR)

AST de l'ASD des expressions

```
ASD([Typedef("expr",
             [Variant("Const",[One("int")]),
              Variant("Var",[One("string")]),
              Variant("Binop",[One("op"),
                               One("expr"),
                               One("expr")])]),
     Typedef("op",
             [Variant("Plus",[]),
              Variant("Minus",[]),
              Variant("Times",[])])])
```

AST de l'ASD des ASD !!!

- l'ASD des ASD est une ASD
- donc l'ASD des ASD est une valeur de l'ASD des ASD

```
ASD([Typedef("asd",
             [Variant("ASD", [List("typedef")])]),
     Typedef("typedef",
             [Variant("Typedef", [One("string"),
                                  List("variant")])
     Typedef("variant",
             [Variant("Variant", [One("string"),
                                  List("argument")
     Typedef("argument",
             [Variant("One", [One("string")]),
              Variant("Optional", [One("string")])
              Variant("List", [One("string")])])])])
```

The End