# Sémantique et Traduction des Langages

## Majeure Sciences et Ingénierie du Logiciel

Marc Pantel

2021 – 2022

# Organisation

- Cours : 10 séances – Marc Pantel
- TD : 8 séances – Marc Pantel/Neeraj Singh
- TP : 9 séances – Marc Pantel/Neeraj Singh
- Mini Projets (20%) en binôme : 2 séances de suivi + 1 test
  Finalisation travaux de TP sur langage fonctionnel miniML
  Finalisation travaux de TP sur langage impératif miniC
- Projet (40%) en quadrinôme : 5 séances de suivi + 1 test
  Extension du langage miniC avec technologies objets miniJava
- Examen (40%) : 1h30 avec documents
- Urgent : Constituer les quadrinômes et binômes associés
- Alternative 1 : Pas de confinement
  - Travaux Dirigées en présenciel
  - Travaux Pratiques en présenciel
- Alternative 2 : Confinement
  - Combinaison TD/TP à distance

# Plan du cours

- Introduction
  - Rappels : Modélisation, Automates et Graphes, GLS
  - Architecture générale
  - Formes de sémantique
- Interprétation
  - Sémantique opérationnelle
  - Sémantique axiomatique
- Compilation
  - Table des Symboles, Arbre abstrait
  - Typage et autres analyses statiques
  - Modèle mémoire, Génération de code
  - Sémantique translationnelle, dénotationnelle
- Vérification de correction

# Rappels

- Modélisation :
  - Structure algébrique des langages
  - Spécification des langages :
    - Expressions régulières,
    - Grammaire (règles de production, EBNF, Conway)
- Automates et Théorie des Langages
  - Automates, Automates à piles, Analyseur descendant récursif
  - Générateurs d'analyseurs lexicaux et syntaxiques
- Ingénierie Dirigée par les Modèles
  - Métamodèles :
    - Représentation abstraite du langage (MOF),
    - Règles de bonne formation (OCL)
  - Syntaxe concrète texte : Xtext

# Principes essentiels

Communication = Echange d'informations

Besoins :
- Représenter les informations possibles
- Reconnaitre une information
- Exploiter une information

Organisation stratifiée : information structurée

Informatique : Science du traitement de l'information

Computer science : Science de la « machine à calculer »

Essentiel :
- Description et manipulation de l'information (langage),
- Traitement d'une information quelconque,
- Traitement d'une manipulation quelconque

D'où :
- Description formelle du langage
- Génération automatique des outils de manipulation

# Références bibliographiques


- Hopcroft, Ullman, Introduction to automata theory, languages and computation, Addison-Wesley, 1979.
- Stern, Fondements mathématiques de l'informatique, McGraw-Hill, 1990.
- Carton, Langages formels, calculabilité et complexité, Vuibert, 2008.
- Aho, Sethi, Ullman, Compilateurs : Principes, Techniques et Outils, InterEditions, 1989.
- Fisher, Leblanc, Crafting a compiler in ADA/in C, Benjamin Cummings, 1991.
- Wilhem, Maurer, Les compilateurs : Théorie, construction, génération, Masson, 1994.
- Appel, Modern Compiler Implementation in Java/ML/C, Cambridge University Press, 1998.
- Winskel, The formal semantics of programming languages : An introduction, MIT Press, 1993.
- Lämmel, Software Languages : Syntax, Semantics and Metaprogramming, Springer (under review), 2017.

# Exemple : fichier /etc/hosts

- Fichier tel qu'il est affiché :

```
#␣Ceci␣est␣un␣commentaire

127.0.0.1       →hal9000␣localhost

#␣En␣voici␣un␣autre

147.127.18.144       →phoenix.enseeiht.fr
```

- Informations brutes : caractères

```
0000000    #  sp   C   e   c   i  sp   e   s   t  sp   u   n  sp   c   o
0000020    m   m   e   n   t   a   i   r   e  nl  nl   1   2   7   .   0
0000040    .   0   .   1  ht   h   a   l   9   0   0   0  sp   l   o   c
0000060    a   l   h   o   s   t  nl  nl   #  sp   E   n  sp   v   o   i
0000100    c   i  sp   u   n  sp   a   u   t   r   e  nl  nl   1   4   7
0000120    .   1   2   7   .   1   8   .   1   4   4  ht   p   h   o   e
0000140    n   i   x   .   e   n   s   e   e   i   h   t   .   f   r  nl
0000160
```

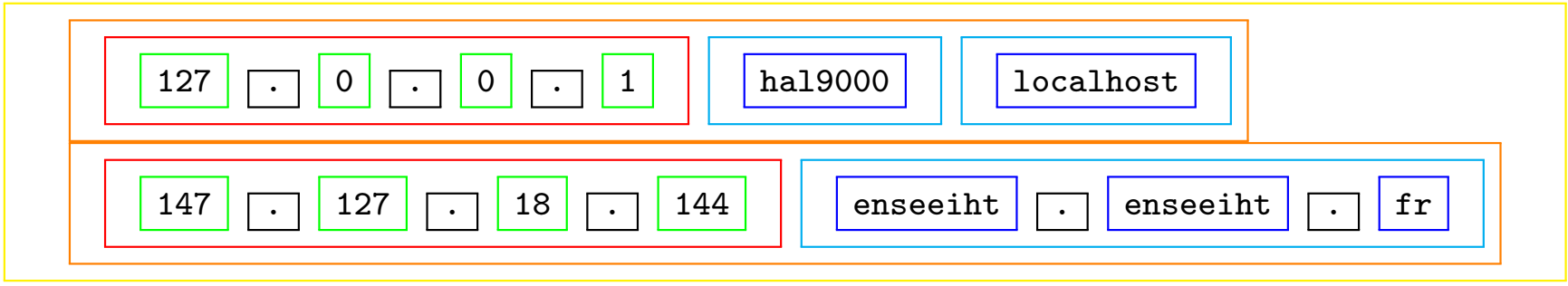
# Analyse lexicale

- Informations élémentaires : commentaire, nombre, identificateur, . (unités lexicales)
- Résultat de l'analyse lexicale :

  `# Ceci est un commentaire` `127` `.` `0` `.` `0` `.` `1`
  `hal9000` `localhost` `# En voici un autre` `147` `.`
  `127` `.` `18` `.` `144` `enseeiht` `.` `enseeiht` `.` `fr`
- Spécification des unités lexicales : Expressions régulières
  - Commentaire : $\#[\hat{}\backslash n]^\star\backslash n$
  - Nombre : $[0-9]^+$
  - Identificateur : $[a-bA-B][a-bA-B0-9]^\star$

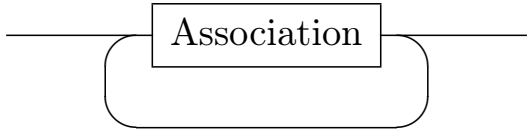
# Analyse syntaxique

- Informations structurées (unités syntaxiques) :
  - Premier niveau : adresse IP, nom qualifié
  - Deuxième niveau : assocation
  - Troisième niveau : document

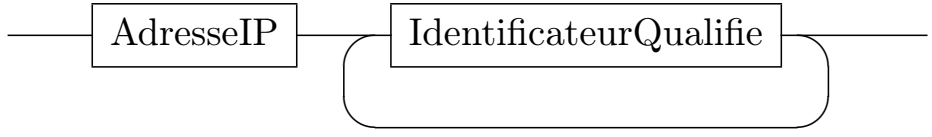



- Spécification des unités syntaxiques : Grammaires (notation de Conway)

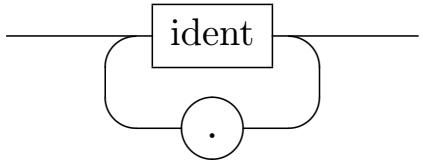
*Document*



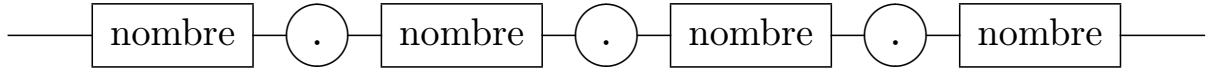

*Association*




*IdentificateurQualifie*

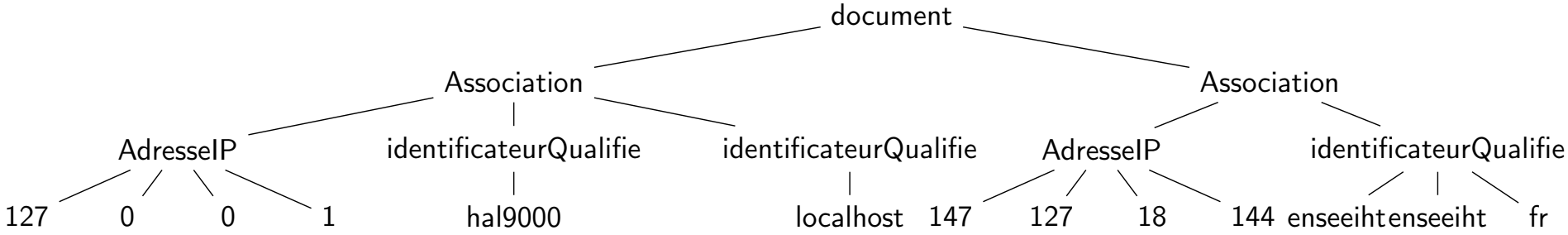



*AdresseIP*

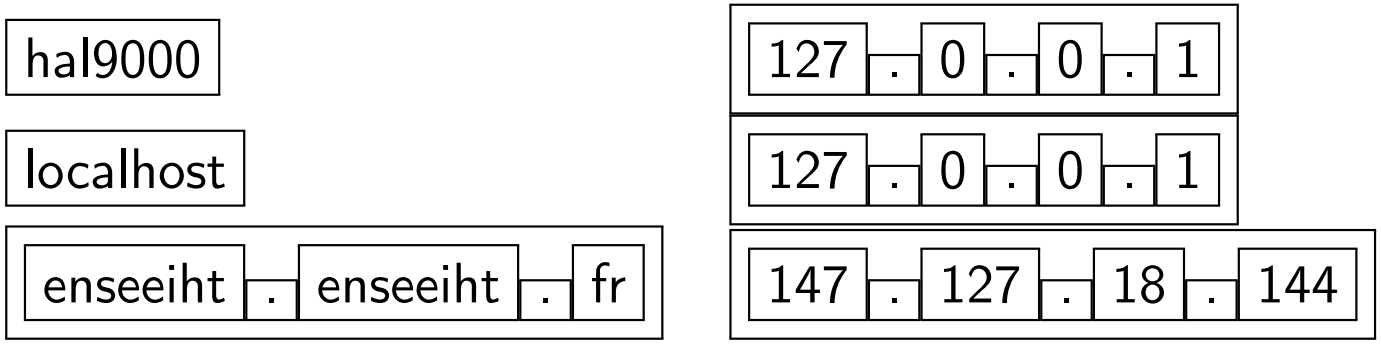


# Analyse sémantique

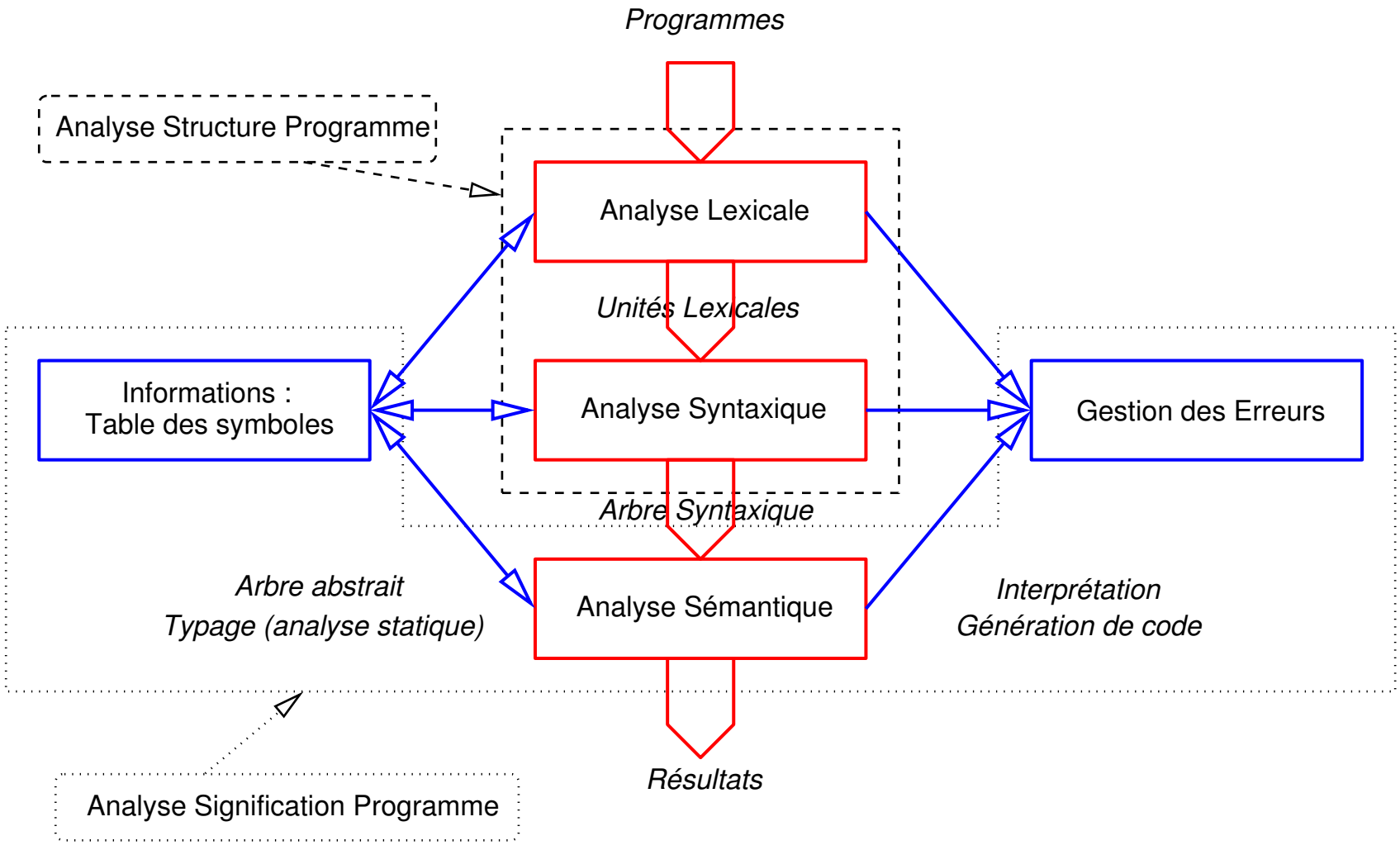
- Structure arborescente associée :



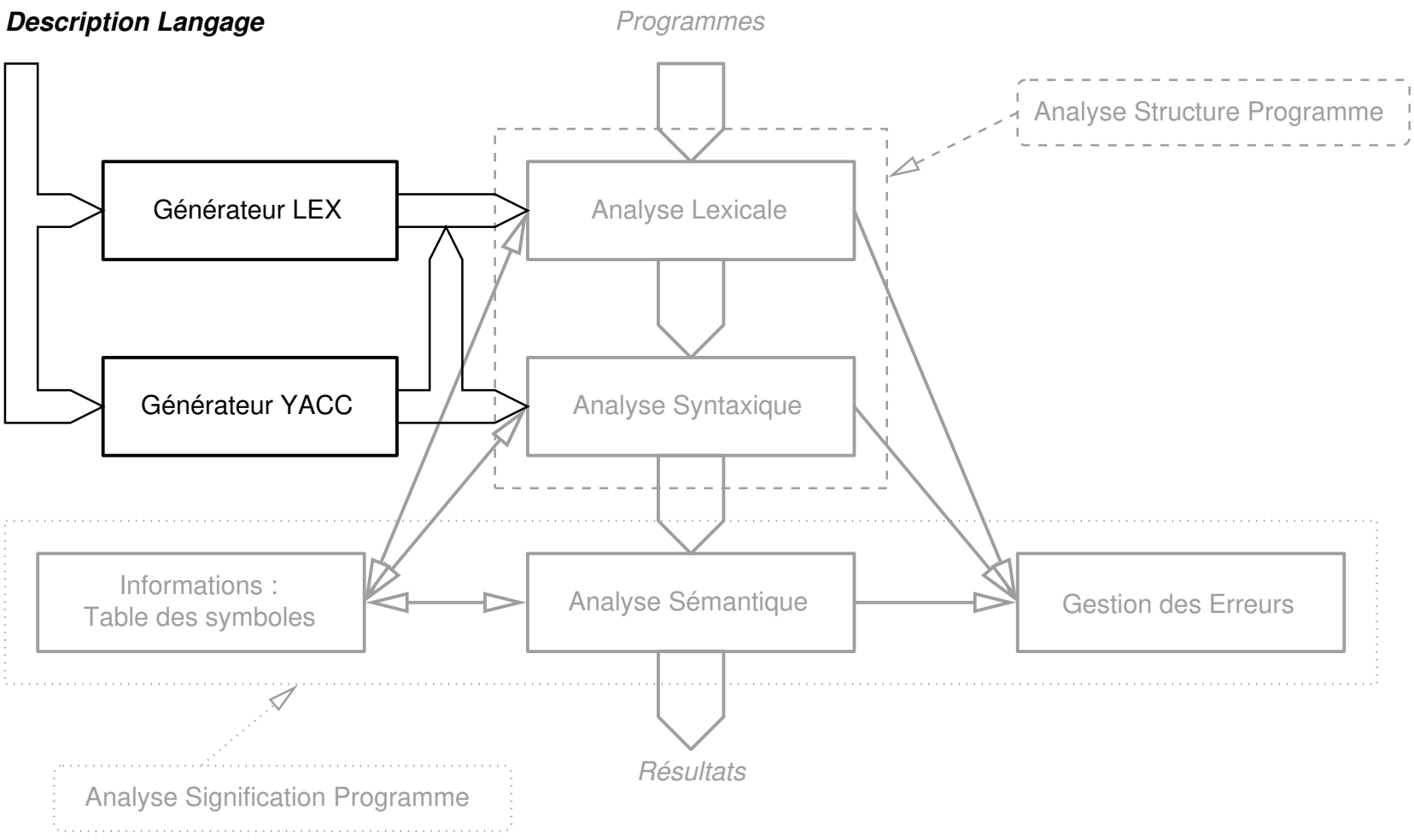

- Exploitation des informations : association nom qualifié/adresse IP (unités sémantiques)



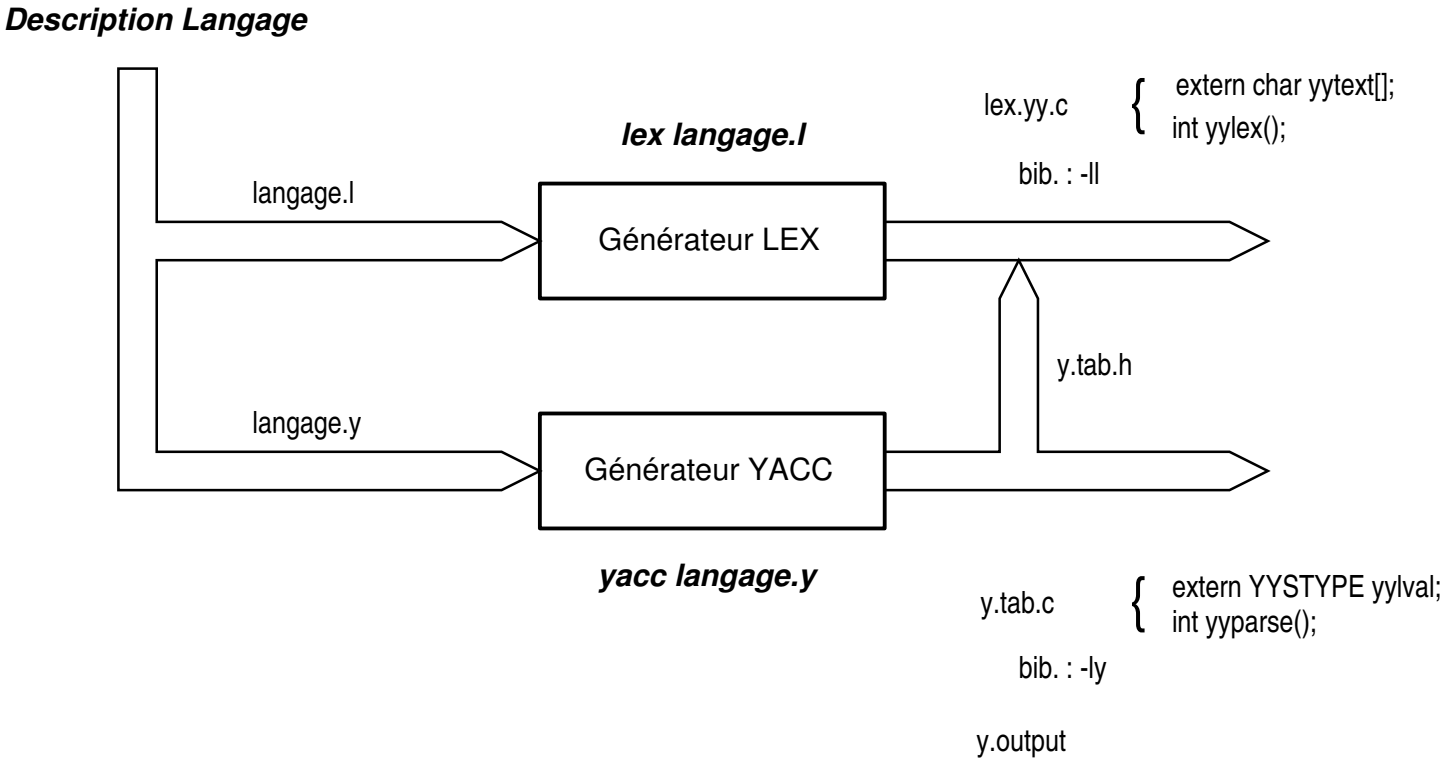
# Structure d'un outil



# Exemple lex et yacc



# Exemple lex et yacc



# Définitions

- Caractère/Symbole : Unité élémentaire d'information
- Unité lexicale (lexème, mot) : Séquence de caractères
- Unité syntaxique (arbre syntaxique, syntème, phrase) : Arbre d'unités lexicales
- Unité sémantique : diverses (arbre abstrait, table des symboles, type, code généré, résultat évaluation, ...)

# Comment organiser les informations ?

- Objectif : Exploitation des informations
- Règle : Choisir le bon niveau de précision
- Unité lexicale : Bloc élémentaire d'information pertinente
- Unité syntaxique : Elément structurant de l'information

# Sémantique formelle des langages

- Objectif : Modélisation la sémantique avec des outils mathématique
- Atteindre la qualité de la modélisation de la syntaxe
- Etudier la cohérence et la complétude
- Prouver la correction des outils
- Générer automatiquement les outils
- Différentes formes :
  - Sémantique opérationnelle : Mécanisme d'exécution des programmes
  - Sémantique axiomatique : Mécanisme de vérification des programmes
  - Sémantique translationnelle : Traduction vers un autre langage équipé d'une sémantique formelle
  - Sémantique dénotationnelle : Traduction vers un formalisme mathématique
- Validation des sémantique par étude équivalence entre formes

# miniML

Trois étapes :

- Expressions : sans effets de bord, similaire dans tous les langages
- Partie fonctionnelle : sans effets de bord
  - Stratégie d'appel par valeur (Call By Value) : Evaluation des paramètres avant appel de fonction
  - Stratégie d'appel par nom (Call By Name) : Evaluation des paramètres lors de l'utilisation dans la fonction
  - Stratégie paresseuse : Appel par nom avec partage des résultats
- Partie Impérative : effets de bord, y compris dans les expressions et la partie fonctionnelle

# miniML

- Expressions :

$$
\begin{array}{lcl}
\textit{Expr} & \rightarrow & \textit{Ident} \\
 & | & \textit{Const} \\
 & | & \textit{Expr Binaire Expr} \\
 & | & \textit{Unaire Expr} \\
 & | & (\ \textit{Expr}\ ) \\
 & & \\
\textit{Const} & \rightarrow & \textit{entier} \mid \textit{booleen} \\
 & & \\
\textit{Unaire} & \rightarrow & - \\
 & & \\
\textit{Binaire} & \rightarrow & + \mid - \mid * \mid / \mid \% \mid \& \mid \mid \\
 & | & == \mid != \mid < \mid <= \mid > \mid >=
\end{array}
$$

# miniML

- Partie Fonctionnelle :

$$
\begin{array}{rcl}
\mathit{Expr} & \rightarrow & \cdots \\
& | & \texttt{let}\ \mathit{Ident}\ \texttt{=}\ \mathit{Expr}\ \texttt{in}\ \mathit{Expr} \\
& | & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Expr}\ \texttt{else}\ \mathit{Expr} \\
& | & \texttt{fun}\ \mathit{Ident}\ \texttt{->}\ \mathit{Expr} \\
& | & \texttt{(}\ \mathit{Expr}\ \texttt{)}\ \mathit{Expr} \\
& | & \texttt{let rec}\ \mathit{Ident}\ \texttt{=}\ \mathit{Expr}\ \texttt{in}\ \mathit{Expr}
\end{array}
$$

- Partie Impératives :

$$
\begin{array}{rcl}
\mathit{Expr} & \rightarrow & \texttt{ref}\ \mathit{Expr} \\
& | & \texttt{!}\ \mathit{Expr} \\
& | & \mathit{Expr}\ \texttt{:=}\ \mathit{Expr} \\
& | & \mathit{Expr}\ \texttt{;}\ \mathit{Expr} \\
& | & \texttt{while}\ \mathit{Expr}\ \texttt{do}\ \mathit{Expr}\ \texttt{done}
\end{array}
$$

# Interprétation : Principes généraux

- Programme qui exécute un programme (émulateur, machine virtuelle, ...)
- Langage hôte/support : Langage de programmation de l'interprète
- Représenter le programme comme une donnée
- Représenter l'exécution comme des données et algorithmes
  - Résultats de l'exécution (dont intermédiaires)
  - Ne pas oublier les erreurs d'exécution (résultats possibles)
  - Pour chaque construction exécutable du langage, identifier :
    1. Variantes selon résultats intermédiaires
    2. Étapes dans chaque variante
    3. Contraintes entre les étapes

# Représentation des programmes et de l'exécution

- Programmes :
  - Arbres abstraits : Abstraction de l'arbre de dérivation (arbre syntaxique)
  - Structure de graphe (relation définition/utilisation)
    - Approche objet : Métamodèles
    - Approche fonctionnelle : Structure d'arbres + Tables des symboles
- Exécution :
  - Valeurs : Exploiter les types de base du langage hôte (booléen, entier, flottant, caractère, chaîne de caractère, ...)
  - Déclarations : Utilisation d'un dictionnaire (table des symboles)
  - Mémoire : Adresses et Espace de données associé

# Application à miniML

- Arbre abstrait : Analyse du code fourni pour travaux pratiques
- Valeurs :

$$\begin{array}{lcl} \mathit{Valeur} & \rightarrow & \mathit{Const} \\ & | & \bot \end{array}$$

- Algorithme d'exécution : Analyse du code fourni pour travaux pratiques

# Sémantique Opérationnelle

- Objectif : Décrire formellement les mécanismes d'exécution des programmes d'un langage
- Principe :
  - Exploiter la syntaxe du langage
  - Décrire l'exécution comme une transformation des programmes
- Notation : Règles de déduction
  - Soient $J_1, \ldots J_n$ et $J$ des jugements :

| | Notation | Signification |
|---|---|---|
| Déduction | $\dfrac{J_1 \quad J_n}{J}$ | si $J_1$ et ...et $J_n$ sont valides alors $J$ est valide |
| Axiome | $\dfrac{}{J}$ | $J$ est valide |

- Jugement d'exécution à grand pas : $\gamma \vdash e \Downarrow v$
  - $\gamma$ : environnement (association *Ident* / *Valeur*)
  - $e$ : expression (*Expr*)
  - $v$ : valeur (*Valeur*)
  - Grand Pas (Big Step) : Calcul complet de l'expression en une valeur
- Partie haute : Étapes intermédiaires (appels récursifs dans interpréte miniML)
- Partie basse : Construction traitée par la règle

# miniML : Constantes et Accès identificateur

- Constante : Valeur ne change pas

$$\frac{}{\gamma \vdash \mathit{entier} \Downarrow \mathit{entier}}$$

$$\frac{}{\gamma \vdash \mathit{booleen} \Downarrow \mathit{booleen}}$$

- Identificateur : Accès à l'environnement
  - Présent : Transmission valeur associée

$$\frac{x \in \gamma \quad \gamma(x) = v}{\gamma \vdash x \Downarrow v}$$

  - Absent : Cas d'erreur

$$\frac{x \notin \gamma}{\gamma \vdash x \Downarrow \bot_{\mathit{undef}}}$$

# miniML : Opérateur Unaire

- Étape préliminaire : Calcul du paramètre
- Variante 1 : Résultat correct du bon type

$$\frac{\gamma \vdash e \Downarrow v \quad v \neq \bot \quad v \in dom\, op \quad v' = op\, v}{\gamma \vdash op\ e \Downarrow v'}$$

- Variante 2 : Résultat erroné

$$\frac{\gamma \vdash e \Downarrow v \quad v = \bot_c}{\gamma \vdash op\ e \Downarrow \bot_c}$$

- Variante 3 : Résultat correct du mauvais type

$$\frac{\gamma \vdash e \Downarrow v \quad v \neq \bot \quad v \notin dom\, op}{\gamma \vdash op\ e \Downarrow \bot_{type}}$$

# miniML : Opérateur Binaire

- Étapes préliminaires : Calcul des paramètres
- Question : Y a t'il un ordre particulier ?
- En absence d'effets de bord : Non, concurrence/parallélisme possible
- Variante 1 : Résultats corrects du bon type

$$\frac{\begin{array}{ll}\gamma \vdash e_1 \Downarrow v_1 & v_1 \neq \bot \\ \gamma \vdash e_2 \Downarrow v_2 & v_2 \neq \bot\end{array} \quad v_1 \times v_2 \in dom\, op \quad v = v_1\ op\ v_2}{\gamma \vdash e_1\ op\ e_2 \Downarrow v}$$

- Variante 2 : Résultat(s) erroné(s)

$$\frac{\gamma \vdash e_1 \Downarrow v_1 \quad v_1 = \bot_c}{\gamma \vdash e_1\ op\ e_2 \Downarrow \bot_c} \qquad \frac{\gamma \vdash e_2 \Downarrow v_2 \quad v_2 = \bot_c}{\gamma \vdash e_1\ op\ e_2 \Downarrow \bot_c}$$

  - Que se passe t'il si deux erreurs se produisent de natures différentes ?
  - Définir une règle qui explicite ce cas

- Variante 3 : Résultat correct du mauvais type

$$\frac{\gamma \vdash e_1 \Downarrow v_1 \quad \gamma \vdash e_2 \Downarrow v_2 \quad v_1 \neq \bot \quad v_2 \neq \bot \quad v_1 \times v_2 \notin dom\, op}{\gamma \vdash e_1\ op\ e_2 \Downarrow \bot_{type}}$$

# miniML : Opérateur Binaire Droite à Gauche

- Imposons un ordre d'évaluation de droite à gauche (celui de OCaML)
- Variante 1 : Résultats corrects du bon type

$$\frac{\gamma \vdash e_2 \Downarrow v_2 \quad v_2 \neq \bot \qquad \gamma \vdash e_1 \Downarrow v_1 \quad v_1 \neq \bot \qquad v_1 \times v_2 \in dom\, op \quad v = v_1\, op\, v_2}{\gamma \vdash e_1\ op\ e_2 \Downarrow v}$$

Attention : Cette règle n'impose pas d'ordre

- Variante 2 : Résultat(s) erroné(s)

$$\frac{\gamma \vdash e_2 \Downarrow v_2 \quad v_2 = \bot_c}{\gamma \vdash e_1\ op\ e_2 \Downarrow \bot_c} \qquad \frac{\gamma \vdash e_2 \Downarrow v_2 \quad v_2 \neq \bot \qquad \gamma \vdash e_1 \Downarrow v_1 \quad v_1 = \bot_c}{\gamma \vdash e_1\ op\ e_2 \Downarrow \bot_c}$$

  - 2 erreurs ne peuvent plus se produire en même temps

- Variante 3 : Résultat correct du mauvais type

$$\frac{\gamma \vdash e_1 \Downarrow v_1 \quad \gamma \vdash e_2 \Downarrow v_2 \quad v_1 \neq \bot \quad v_2 \neq \bot \quad v_1 \times v_2 \notin dom\, op}{\gamma \vdash e_1\ op\ e_2 \Downarrow \bot_{type}}$$

# Exemple d'exécution d'un programme miniML

- Prenons : $\gamma = \{v \mapsto 2\}$
- Calculons le programme miniML : $1 + v * 3$
- L'arbre est trop volumineux, décomposons en :

$$A = \frac{v \in \gamma \quad \gamma(v) = 2}{\gamma \vdash v \Downarrow 2}$$

$$B = \frac{\begin{array}{ccc} A & \gamma \vdash 3 \Downarrow 3 & \\ 2 \neq \bot & 3 \neq \bot & 2 \times 3 \in dom* \quad 6 = 2 * 3 \end{array}}{\gamma \vdash v * 3 \Downarrow 6}$$

$$\frac{\begin{array}{ccc} \gamma \vdash 1 \Downarrow 1 & B & \\ 1 \neq \bot & 6 \neq \bot & 1 \times 6 \in dom+ \quad 7 = 1 + 6 \end{array}}{\gamma \vdash 1 + v * 3 \Downarrow 7}$$

# Sémantique à petit pas (Small Step)

- Sémantique à grand pas autorise la concurrence mais ne la détaille pas
- Contrainte : passage de l'expression à la valeur en une seule étape
- Sémantique à petit pas : décompose ce passage en étapes microscopiques

$$\frac{\begin{array}{l} v_1 \neq \bot \\ v_2 \neq \bot \end{array} \qquad v_1 \times v_2 \in dom\, op \quad v = v_1\ op\ v_2}{\gamma \vdash\ v_1\ op\ v_2\ \Rightarrow\ v}$$

- Explicite l'entrelacement des micros-étapes

$$\frac{\begin{array}{c} \gamma \vdash\ e_1\ \Rightarrow\ e_3 \\ \gamma \vdash\ e_2\ \Rightarrow\ e_4 \end{array}}{\gamma \vdash\ e_1\ op\ e_2\ \Rightarrow\ e_3\ op\ e_4}$$

# Définitions récursives

- Syntaxe : $\texttt{let rec } f = e_1 \texttt{ in } e_2$
- Rappel : Définition simple

$$\frac{\gamma \vdash e_1 \Downarrow v_1 \quad \gamma :: \{x \mapsto v_1\} \vdash e_2 \Downarrow v_2}{\gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 \Downarrow v_2}$$

- Rendons $f$ visible dans $e_1$ :

$$\frac{\gamma :: \{f \mapsto v_1\} \vdash e_1 \Downarrow v_1 \quad \gamma :: \{f \mapsto v_1\} \vdash e_2 \Downarrow v_2}{\gamma \vdash \texttt{let rec } f = e_1 \texttt{ in } e_2 \Downarrow v_2}$$

- Question : Est ce bien fondé ?
- Remarque :
  $\texttt{let rec } f = e_1 \texttt{ in } e_2 \equiv \texttt{let } f = \texttt{let rec } f = e_1 \texttt{ in } e_1 \texttt{ in } e_2$
- Exploitons cette relation :

$$\frac{\gamma \vdash \texttt{let rec } f = e_1 \texttt{ in } e_1 \Downarrow v_1 \quad \gamma :: \{f \mapsto v_1\} \vdash e_2 \Downarrow v_2}{\gamma \vdash \texttt{let rec } f = e_1 \texttt{ in } e_2 \Downarrow v_2}$$

- Est ce un progrès ?

# Définitions récursives

- Si nous le faisons une seconde fois :

$$\frac{\gamma \vdash \texttt{let rec } f = e_1 \texttt{ in } e_1 \Downarrow v_1 \quad \gamma :: \{f \mapsto v_1\} \vdash e_1 \Downarrow v_1}{\gamma \vdash \texttt{let rec } f = e_1 \texttt{ in } e_1 \Downarrow v_1}$$

- Si $e_1$ s'évalue en une fonction $\langle\, \texttt{fun } x \texttt{ -> } e_3 \,,\, \gamma_{def} \,\rangle$
- Nous pouvons alors poursuivre le calcul de $e_2$ en exploitant cette fermeture
- Nous en déduisons la règle simplifiée dans laquelle nous gelons le calcul de la définition récursive

$$\frac{\gamma :: \{f \mapsto \langle\, \texttt{let rec } f = e_1 \texttt{ in } e_1 \,,\, \gamma \,\rangle\} \vdash e_2 \Downarrow v}{\gamma \vdash \texttt{let rec } f = e_1 \texttt{ in } e_2 \Downarrow v}$$

- Il faut alors ajouter une règle qui degèle le calcul lors de l'accès à $f$ dans l'environnement :

$$\frac{x \in \gamma \quad \gamma(x) = \langle\, e \,,\, \gamma_{def} \,\rangle \quad \gamma_{def} \vdash e \Downarrow v}{\gamma \vdash x \Downarrow v}$$

$$\frac{x \in \gamma \quad \gamma(x) = v \quad v \neq \langle\, e \,,\, \gamma_{def} \,\rangle}{\gamma \vdash x \Downarrow v}$$

# Analyse de programmes

- Objectif : Déterminer les propriétés des programmes
- Analyse dynamique : Exécuter les programmes pour observer les propriétés
- Approche incomplète :
  - Exécution finie : Nombre d'étapes d'exécution fini
  - Nombre d'exécution fini
- Analyse statique : Déterminer les propriétés sans exécuter les programmes
  - Abstraction finie d'une exécution
  - Exécution symbolique du programme (interprétation abstraite)
  - Approche complète : Abstraction de toutes étapes de toutes les exécutions possibles
  - Approche correcte : Sur-approximation des propriétés réelles
- Exemple : Détecter certaines erreurs d'exécution sans exécuter les programmes (Définitions, Typage, Erreurs de calcul, Consommation ressources, ...)

# Mécanisme de typage

- Notion de type : Ensemble de valeurs pour lesquelles le programme a le même comportement
- Langage des types possibles : Syntaxe des types
  - Pour les expressions de miniML : `bool` et `int`
- Sémantique des types : Ensemble des valeurs possibles y compris les erreurs à l'exécution
  - $[\![\texttt{bool}]\!] = \{\texttt{true}, \texttt{false}, \bot_{runtime}\}$
  - $[\![\texttt{int}]\!] = \mathbb{Z} \cup \{\bot_{runtime}\}$
- Relations de comparaison des types :
  - Égalité
  - Compatibilité :
    - Généricité/Instanciation, Polymorphisme paramètrique
    - Sous-typage, Polymorphisme d'héritage
    - ...
- Règles de calcul (en miniML, Unification des types)

# Analyseur statique : Principes généraux

- Programme qui détermine les propriétés d'un programme
- Langage hôte/support : Langage de programmation de l'analyseur
- Représenter le programme comme une donnée
- Représenter les propriétés comme des données
- Exprimer les règles de vérification comme des algorithmes
  - Résultats de la vérification (dont intermédiaires)
  - Pour chaque construction exécutable du langage, identifier :
    1. Variantes selon résultats intermédiaires
    2. Étapes dans chaque variante
    3. Contraintes entre les étapes

# Application à miniML

- Arbre abstrait : voir vidéo séparée
- Syntaxe des types :

$$\begin{array}{rcl} \mathit{Type} & \rightarrow & \texttt{bool} \\ & | & \texttt{int} \end{array}$$

- Représentation des types et unification : voir vidéo séparée
- Algorithme de typage : voir vidéo séparée

# Sémantique Axiomatique

- Objectif : Décrire formellement les mécanismes d'analyse des propriétés des programmes d'un langage
- Principe :
  - Exploiter la syntaxe du langage
  - Décrire les relations entre les constructions du langage et les propriétés
- Notation : Règles de déduction
- Jugement de typage : $\sigma \vdash e : \tau$
  - $\sigma$ : environnement (association *Ident* / *Type*)
  - $e$ : expression (*Expr*)
  - $\tau$ : type (*Type*)
- Partie haute : Étapes intermédiaires (appels récursifs dans typeur miniML)
- Partie basse : Construction traitée par la règle
- Principe de construction : Règles d'exécution congrue par la sémantique des types (façon classes d'équivalence)

# miniML : Constantes et Accès identificateur

- Règles d'évaluation :

$$\frac{}{\gamma \vdash \mathit{entier} \Downarrow \mathit{entier}} \qquad \frac{}{\gamma \vdash \mathit{booleen} \Downarrow \mathit{booleen}}$$

- Règles de typage :

$$\frac{}{\sigma \vdash \mathit{entier} : \texttt{int}} \qquad \frac{}{\sigma \vdash \mathit{booleen} : \texttt{bool}} \qquad \forall \tau, \frac{}{\sigma \vdash \bot_{\mathit{runtime}} : \tau}$$

- Identificateur : Accès à l'environnement
  - Transmission valeur associée :

$$\frac{x \in \gamma \quad \gamma(x) = v}{\gamma \vdash x \Downarrow v}$$

  - Règle de typage associée :

$$\frac{x \in \sigma \quad \sigma(x) = \tau}{\sigma \vdash x : \tau}$$

# miniML : Opérateur Unaire

- Étape préliminaire : Traitement du paramètre
- Variante 1 : Résultat correct du bon type

$$\frac{\gamma \vdash e \Downarrow v \quad v \neq \bot \quad v \in dom\, op \quad v' = op\, v}{\gamma \vdash op\, e \Downarrow v'}$$

- Variante 2 : Résultat erroné

$$\frac{\gamma \vdash e \Downarrow v \quad v = \bot_c}{\gamma \vdash op\, e \Downarrow \bot_c} \text{ avec } c \neq type$$

- Règle de typage associée :

$$\frac{\sigma \vdash e : \tau \quad \tau = dom\, op \quad \tau' = codom\, op}{\sigma \vdash op\, e : \tau'}$$

- Notons que : $\bot_c \in [\![\tau]\!] \wedge \bot_c \in [\![\tau']\!]$ avec $c \neq type$

# miniML : Opérateur Binaire Droite à Gauche

- Étapes préliminaires : Traitement des paramètres
- Variante 1 : Résultats corrects du bon type

$$\frac{\begin{array}{cc}\gamma \vdash e_2 \Downarrow v_2 & \gamma \vdash e_1 \Downarrow v_1 \\ v_2 \neq \bot & v_1 \neq \bot\end{array} \qquad v_1 \times v_2 \in dom\ op \quad v = v_1\ op\ v_2}{\gamma \vdash e_1\ op\ e_2 \Downarrow v}$$

- Variante 2 : Résultat(s) erroné(s) (avec $c \neq type$)

$$\frac{\gamma \vdash e_2 \Downarrow v_2 \quad v_2 = \bot_c}{\gamma \vdash e_1\ op\ e_2 \Downarrow \bot_c} \qquad \frac{\begin{array}{c}\gamma \vdash e_2 \Downarrow v_2 \\ v_2 \neq \bot\end{array} \qquad \gamma \vdash e_1 \Downarrow v_1 \quad v_1 = \bot_c}{\gamma \vdash e_1\ op\ e_2 \Downarrow \bot_c}$$

- Règle de typage associée :

$$\frac{\sigma \vdash e_1 : \tau_1 \quad \sigma \vdash e_2 : \tau_2 \quad \tau_1 \times \tau_2 = dom\ op \quad \tau = codom\ op}{\sigma \vdash e_1\ op\ e_2 : \tau}$$

- Notons que : $\bot_c \in [\![\tau_1]\!] \wedge \bot_c \in [\![\tau_2]\!] \wedge \bot_c \in [\![\tau]\!]$ avec $c \neq type$

# Exemple de typage d'un programme miniML

- Prenons : $\sigma = \{v : \texttt{int}\}$
- Typons le programme miniML : $1 + v * 3$
- L'arbre est trop volumineux, décomposons en :

$$A = \frac{v \in \sigma \quad \sigma(v) = \texttt{int}}{\sigma \vdash v : \texttt{int}}$$

$$B = \frac{\begin{array}{cc} A & \texttt{int} \times \texttt{int} = dom* \\ \sigma \vdash 3 : \texttt{int} & \texttt{int} = codom* \end{array}}{\sigma \vdash v * 3 : \texttt{int}}$$

$$\frac{\begin{array}{cc} \sigma \vdash 1 : \texttt{int} & \texttt{int} \times \texttt{int} = dom+ \\ B & \texttt{int} = codom+ \end{array}}{\sigma \vdash 1 + v * 3 : \texttt{int}}$$

# Typage et Fonctions

- $\tau_P \to \tau_R$ : type d'une fonction dont
  - le paramètre est de type $\tau_P$
  - le résultat est de type $\tau_R$
- Sémantique :
  $[\![\tau_P \to \tau_R]\!] = \{\langle\, e_F\,,\,\gamma\,\rangle \mid \gamma \vdash (\, e_F\,)\ v_P \Downarrow v_R, v_P \in [\![\tau_P]\!], v_R \in [\![\tau_R]\!]\}$
- Ensemble des expressions dont le comportement est une fonction dont le résultat est dans $\tau_R$ si le paramètre est dans $\tau_P$
- Notation (fermeture) : $\langle\, e_F\,,\,\gamma\,\rangle$ représente une expression $e_F$ et son environnement d'évaluation $\gamma$

# Unification

- Résolution de contraintes d'égalité entre termes (ici les types) $\{\tau_i = \tau_i'\}$
- Détermine si le système possède des solutions
- Décomposition pour mettre en forme normale $\{\alpha_i = \tau_i\}$ sans cycle $\alpha_i \notin FV(\tau_i)$
- Exemple : $\alpha_1 \to \tau_2 = \tau_1 \to \alpha_2$ est décomposée en $\{\alpha_1 = \tau_1, \alpha_2 = \tau_2\}$
- Construit la fermeture reflexive, symetrique et transitive de $=$
- Détermine les valeurs des variables pour que les contraintes soient satisfaites
- Implantation efficace à base de références pour ne pas effectuer de substitutions

# Correction du typage par rapport à l'exécution

- Correction du typage : Si une expression est bien typée alors elle ne s'évalue pas en erreur de type

$$\forall e \in \mathcal{L}_{miniML}, \forall \tau, \vdash e : \tau \rightarrow \neg \vdash e \Downarrow \bot_{type} \wedge \neg \vdash e \Downarrow \bot_{undef}$$

- Lemme de continuité (Subject Reduction) : Pas élémentaire d'évaluation bien typée

$$\begin{array}{l} \forall e \in \mathcal{L}_{miniML}, \\ \forall v \in \mathcal{V}_{miniML}, \\ \forall \tau, \forall \gamma, \forall \sigma, \end{array} \left\{ \begin{array}{l} dom\,\gamma = dom\,\sigma \\ \sigma \vdash e : \tau \\ \gamma \vdash e \Downarrow v \\ \forall x \in \gamma, \vdash \gamma(x) : \sigma(x) \end{array} \right. \rightarrow \sigma \vdash v : \tau$$

- Preuve du lemme par induction sur la structure du langage
- Trivial par construction des règles de typage à partir des règles d'évaluation
- Preuve du théoreme de correction : Par l'absurde en utilisant le lemme et le fait que l'erreur de typage n'est pas bien typée

# Preuve de continuité : Constante

- Cas $e = \mathit{entier}$ :

$$\forall v \in \mathcal{V}_{\mathit{miniML}}, \forall \tau, \forall \gamma, \forall \sigma, \begin{cases} \mathit{dom}\,\gamma = \mathit{dom}\,\sigma \\ \sigma \vdash \mathit{entier} : \tau \\ \gamma \vdash \mathit{entier} \Downarrow v \\ \forall x \in \gamma, \vdash \gamma(x) : \sigma(x) \end{cases} \rightarrow \sigma \vdash v : \tau$$

- Hypothèses :
  $\sigma \vdash \mathit{entier} : \tau \wedge \gamma \vdash \mathit{entier} \Downarrow v \wedge \forall x \in \gamma, \vdash \gamma(x) : \sigma(x)$
- Application des règles d'évaluation et de typage :

$$\frac{}{\gamma \vdash \mathit{entier} \Downarrow \mathit{entier}} \quad \frac{}{\sigma \vdash \mathit{entier} : \texttt{int}}$$

- Donc $v = \mathit{entier}$ et $\tau = \mathit{int}$ et $\forall \sigma, \sigma \vdash \mathit{entier} : \texttt{int}$, CQFD.
- Idem pour les booléens

# Preuve de continuité : Variable

- Cas $e = x$ :

$$\forall v \in \mathcal{V}_{miniML}, \forall \tau, \forall \gamma, \forall \sigma, \begin{cases} dom\,\gamma = dom\,\sigma \\ \sigma \vdash x : \tau \\ \gamma \vdash x \Downarrow v \\ \forall x \in \gamma, \vdash \gamma(x) : \sigma(x) \end{cases} \rightarrow \sigma \vdash v : \tau$$

- Hypothèses : $\sigma \vdash x : \tau \wedge \gamma \vdash x \Downarrow v \wedge \forall x \in \gamma, \vdash \gamma(x) : \sigma(x)$
- Application des règles d'évaluation et de typage :

$$\frac{x \in \gamma \quad \gamma(x) = v}{\gamma \vdash x \Downarrow v} \qquad \frac{x \in \sigma \quad \sigma(x) = \tau}{\sigma \vdash x : \tau}$$

- Nous avons $\sigma \vdash x : \tau$ donc $x \in \sigma$ donc $x \in \gamma$ donc $\gamma \vdash x \Downarrow v$ et $\vdash \gamma(x) : \sigma(x)$, CQFD.

# Preuve de continuité : Opérateur unaire

- Cas $e = op\ e'$ :

$$\forall v \in \mathcal{V}_{miniML}, \forall \tau, \forall \gamma, \forall \sigma, \begin{cases} dom\,\gamma = dom\,\sigma \\ \sigma \vdash op\ e' : \tau \\ \gamma \vdash op\ e' \Downarrow v \\ \forall x \in \gamma, \vdash \gamma(x) : \sigma(x) \end{cases} \rightarrow \sigma \vdash v : \tau$$

- Hypothèse d'induction sur $e'$

$$\forall v' \in \mathcal{V}_{miniML}, \forall \tau', \forall \gamma', \forall \sigma', \begin{cases} dom\,\gamma' = dom\,\sigma' \\ \sigma' \vdash e' : \tau' \\ \gamma' \vdash e' \Downarrow v' \\ \forall x \in \gamma', \vdash \gamma'(x) : \sigma'(x) \end{cases} \rightarrow \sigma' \vdash v' : \tau'$$

# Preuve de continuité : Opérateur unaire

- Hypothèses :
  $\sigma \vdash op\ e' : \tau \wedge \gamma \vdash op\ e' \Downarrow v \wedge \forall x \in \gamma, \vdash \gamma(x) : \sigma(x)$
- Application des règles d'évaluation :

$$\frac{\gamma \vdash e' \Downarrow v' \quad v' \neq \bot \quad v' \in dom\, op \quad v = op\, v'}{\gamma \vdash op\ e' \Downarrow v}$$

$$\frac{\gamma \vdash e' \Downarrow v' \quad v' = \bot_c}{\gamma \vdash op\ e' \Downarrow \bot_c} \text{ avec } c \notin \{type, undef\}$$

- Application des règles de typage :

$$\frac{\sigma \vdash e' : \tau' \quad \tau' = dom\, op \quad \tau = codom\, op}{\sigma \vdash op\ e' : \tau}$$

- Nous pouvons appliquer l'hypothèse d'induction et typer $v = op\, v'$ ou $v = \bot_c$, CQFD.

# Cours, TD, TP, mini-projet : miniC

- Types de données :
  - Types de base : `boolean`, `int`, `char`, `string`
  - Types structurées : $n$-uplets, Tableaux, Pointeurs, Enregistrements, Déclaration de types
- Algorithmes
  - Expressions sans effets de bord (sauf affectation)
  - Instructions : séquence, conditionnelle, répétition
  - Déclarations de variables avec et sans initialisation
  - Fonctions, Procédures avec récursivité

# Projet : miniJava

- Classes et Interfaces génériques avec Instanciation explicite
- Constructeurs, Attributs et Méthodes d'Instances et de Classes avec droits d'accès et restriction d'héritage
- Polymorphisme d'héritage et Liaison tardive

# Grammaires Attribuées : Principes généraux

- Objectif : Enricher la spécification de la syntaxe avec des éléments de sémantique
- Support : Règles de production
- Attributs sémantiques : Informations typées associées aux symboles (terminaux, non-terminaux)
- Équations sémantique : Relations entre les attributs des symboles d'une règle de production
- Question : Pour un programme donné, est il possible de calculer les valeurs des attributs sémantiques ?
- Solution : Calcul d'un point fixe sur les équations sémantiques
- Problème : Existence du point fixe en temps fini ? Raisonnable ?
- Approche : Restriction sur la forme des équations pour assurer la terminaison

# Grammaires attribuées : Méthode

- Identifier les informations :
  - Disponibles avant l'analyse du programme : Contexte de l'analyse
  - Associées aux terminaux du programme : Informations lexicales
  - Associées à la structure de l'arbre de dérivation : Informations syntaxiques
  - Résultant de l'analyse sémantique
- Choisir des exemples représentatifs du langage
- Étiqueter :
  - Racine de l'arbre (axiome de la grammaire) : Informations de contexte
  - Feuilles de l'arbre (unités lexicales) : Informations lexicales
  - NœUds de l'arbre : Informations syntaxiques
- Étiqueter la racine avec les résultats attendus
- Identifier les relations entre les résultats attendus et les informations disponibles
- Introduire les attributs nécessaires pour les nœuds intermédiaires
- Définir et placer les actions sémantiques pour chaque nœud

# Exemple : Evaluation de miniML

- Attribut : Environnement d'évaluation (associe une valeur à chaque variable)
- Attribut : Valeur de l'expression
- Action sémantique : Régle d'évaluation associée à la régle de production

$$
\begin{array}{lcl}
E & \rightarrow & E_1 + E_2 \\
 & & E_1.env = E.env \\
 & & E_2.env = E.env \\
 & & E.value = Compute(+, E_1.value, E_2.value)
\end{array}
$$

$$
\begin{array}{lcl}
E & \rightarrow & E_1 * E_2 \\
 & & E_1.env = E.env \\
 & & E_2.env = E.env \\
 & & E.value = Compute(*, E_1.value, E_2.value)
\end{array}
$$

$$
\begin{array}{lcl}
E & \rightarrow & ( E_1 ) \\
 & & E_1.env = E.env \\
 & & E.value = E_1.value
\end{array}
$$

$$
\begin{array}{lcl}
E & \rightarrow & - E_1 \\
 & & E_1.env = E.env \\
 & & E.value = Compute(-, E_1.value)
\end{array}
$$

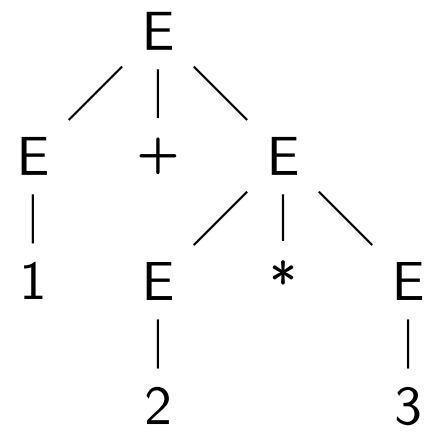
$$
\begin{array}{lcl}
E & \rightarrow & \texttt{entier} \\
 & & E.value = \texttt{entier}.value
\end{array}
$$

$$
\begin{array}{lcl}
E & \rightarrow & \texttt{ident} \\
 & & E.value = LookUp(E.env, \texttt{ident}.value)
\end{array}
$$

# Rappel : Arbres de dérivation (syntaxique)

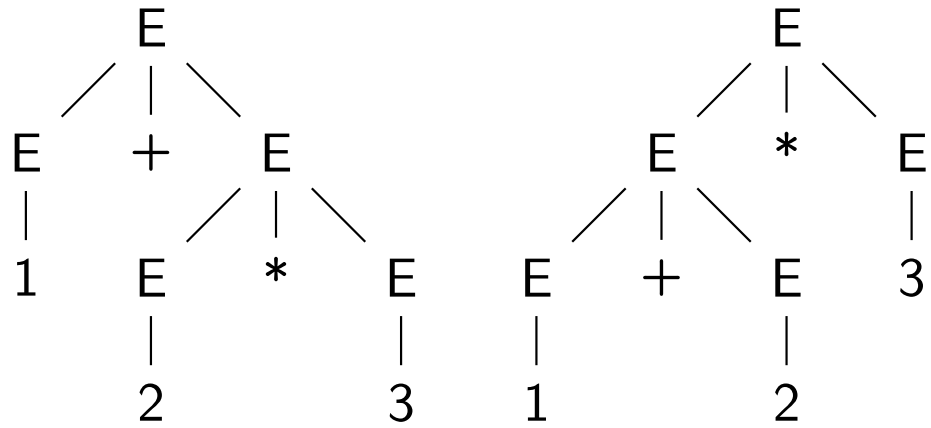
- Objectif : Représenter la structure du mot induite par les règles de production lors d'une dérivation
- Feuilles de l'arbre : Terminaux composant le mot
- Racine de l'arbre : Axiome
- Nœuds de l'arbre : Non-terminaux apparaissant dans la dérivation
- Branches de l'arbre : Règles de production

Exemple :
Arbre de dérivation pour `1 + 2 * 3`



# Rappel : Grammaire et langage ambigu

- Une grammaire est ambigue s'il existe plusieurs arbres de dérivation distincts pour un même mot
- Exemple : Arbres de dérivation pour `1 + 2 * 3`



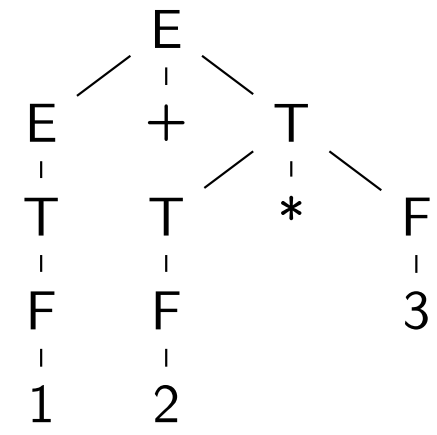

- Un langage est ambigu si toutes les grammaires le représentant sont ambigues

# Grammaires pour les expressions

- Associativité codée par récursivité
- Priorité codée par imbrication des régles
- Grammaire $LR(k)$ :

$$
\begin{array}{llll}
E & \to & E + T & F \to (\ E\ ) \\
E & \to & T & F \to -\ F \\
T & \to & T * F & F \to \texttt{entier} \\
T & \to & F & F \to \texttt{ident}
\end{array}
$$

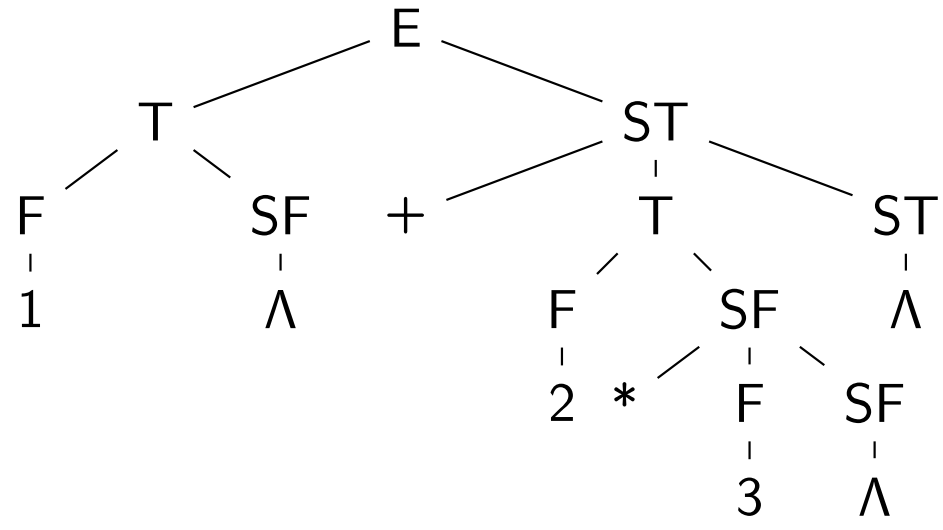
Arbre associé :




- Grammaire $LL(k)$ :

$$
\begin{array}{lll}
E & \to & T\ ST \\
ST & \to & +\ T\ ST \\
ST & \to & \Lambda \\
T & \to & F\ SF \\
SF & \to & *\ F\ SF \\
SF & \to & \Lambda
\end{array}
$$

Arbre associé :



# Grammaires L-attribuées

- Objectif : Calcul pendant l'analyse syntaxique
- Hypothèse : Parcours de l'arbre de dérivation descendant puis ascendant de gauche à droite
- Remarque : Compatible avec analyse descendante récursive (grammaires $LL(k)$)
- Nature des attributs sémantiques des non terminaux :
  - Hérité (parcours descendant) : Calculé avant l'analyse du non terminal
  - Synthétisé (parcours ascendant) : Calculé pendant l'analyse du non terminal
- Forme des équations : Fonctions qui calculent la valeur des attributs
  - Synthétisés du symbole non terminal associé à la règle
  - Hérités des symboles non terminaux exploités par la règle
- Contrainte : Incompatible avec l'analyse ascendante (grammaires $LR(k)$)

# Grammaires S-attribuées

- Objectif : Compatible avec analyseurs ascendants
- Uniquement des Attributs synthétisés
- Exécution des équations en fin de règle de production
- Exemples d'outils : `ocamlyacc`, `menhir`
- Qu'en est il des outils classiques de la famille `yacc` et `bison` ?
  - Utilisation de variables globales pour émuler les attributs hérités
  - Ajout de non-terminaux virtuels pour les actions sémantiques internes aux règles : Exécution de l'action sur la réduction de la règle
  - Introduit des conflits qui imposent la factorisation des règles : sous-ensemble des grammaires $LR(k)$
- Problèmes : Restrictions trop fortes pour la plupart des sémantiques
- Méthode associée :
  - Construction de l'arbre abstrait
  - Parcours de l'arbre abstrait pour les sémantiques plus complexes

# Exemple : Evaluation de miniML

- Attribut hérité : Environnement d'évaluation
- Attribut synthétisé : Valeur de l'expression
- Action sémantique : Régle d'évaluation associée à la régle de production
- Grammaire L-attribuée

$$
\begin{array}{lcl}
E & \rightarrow & \#1\ E + \#2\ T\ \#3 \\
\#1 & : & E_1.env = E.env \\
\#2 & : & T.env = E.env \\
\#3 & : & E.value = Compute(+, E_1.value, T.value)
\end{array}
$$

$$
\begin{array}{lcl}
E & \rightarrow & \#1\ T\ \#2 \\
\#1 & : & T.env = E.env \\
\#2 & : & E.value = T.value
\end{array}
$$

$$
\begin{array}{lcl}
T & \rightarrow & \#1\ T * \#2\ F\ \#3 \\
\#1 & : & T_1.env = T.env \\
\#2 & : & F.env = T.env \\
\#3 & : & E.value = Compute(*, T_1.value, F.value)
\end{array}
$$

$$
\begin{array}{lcl}
T & \rightarrow & \#1\ F\ \#2 \\
\#1 & : & F.env = T.env \\
\#2 & : & T.value = F.value
\end{array}
$$

$$
\begin{array}{lcl}
F & \rightarrow & \#1\ (\ E\ )\ \#2 \\
\#1 & : & E.env = F.env \\
\#2 & : & F.value = E.value
\end{array}
$$

$$
\begin{array}{lcl}
F & \rightarrow & \#1\ -F\ \#2 \\
\#1 & : & F_1.env = F.env \\
\#2 & : & F.value = Compute(-, F_1.value)
\end{array}
$$

$$
\begin{array}{lcl}
F & \rightarrow & \texttt{entier}\ \#1 \\
\#1 & : & F.value = \texttt{int}
\end{array}
$$

$$
\begin{array}{lcl}
F & \rightarrow & \texttt{ident}\ \#1 \\
\#1 & : & F.value = LookUp(F.env, \texttt{ident}.value)
\end{array}
$$

# Exemple : Evaluation de miniML

- Élimination de la récursivité à gauche
- Attributs hérités supplémentaires : `inhValue`

$$
\begin{array}{lcl}
E & \rightarrow & \#1\ T\ \#2\ ST\ \#3 \\
\#1 & : & T.env = E.env \\
\#2 & : & ST.env = E.env \\
 & & ST.inhValue = T.value \\
\#3 & : & E.value = ST.value
\end{array}
$$

$$
\begin{array}{lcl}
ST & \rightarrow & \#1\ +\ T\ \#2\ ST\ \#3 \\
\#1 & : & ST_1.env = ST.env \\
\#2 & : & T.env = ST.env \\
 & & ST_1.inhValue = Compute(+, ST.inhValue, T.value) \\
\#3 & : & ST.value = ST_1.value
\end{array}
$$

$$
\begin{array}{lcl}
ST & \rightarrow & \#1 \\
\#1 & : & ST.value = ST.inhValue
\end{array}
$$

$$
\begin{array}{lcl}
T & \rightarrow & \#1\ F\ \#2\ SF\ \#3 \\
\#1 & : & F.env = T.env \\
\#2 & : & SF.env = T.env \\
 & & SF.inhValue = F.value \\
\#3 & : & T.value = SF.value
\end{array}
$$

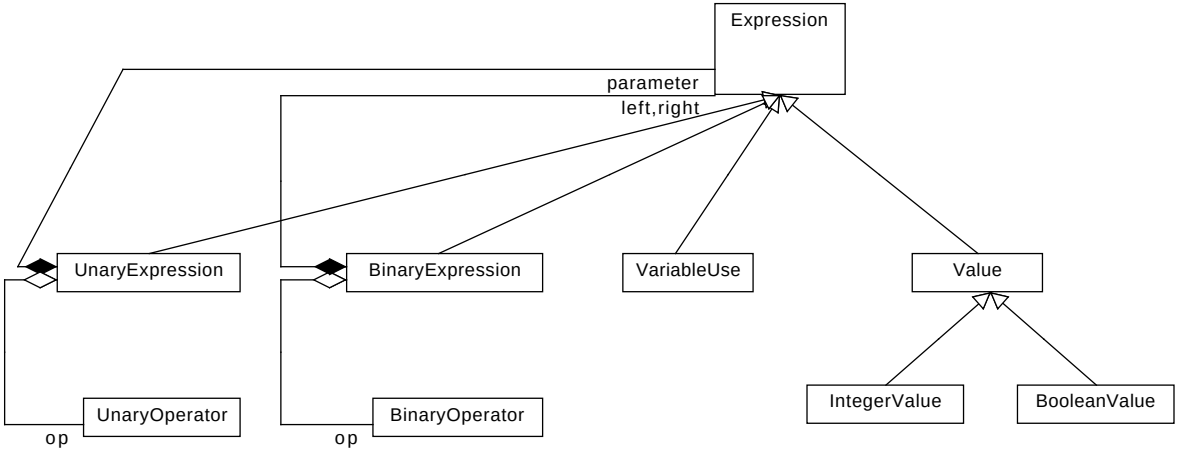
$$
\begin{array}{lcl}
SF & \rightarrow & \#1\ *\ F\ \#2\ SF\ \#3 \\
\#1 & : & SF_1.env = SF.env \\
\#2 & : & F.env = SF.env \\
 & & SF_1.inhValue = Compute(*, SF.inhValue, F.value) \\
\#3 & : & SF.value = SF_1.value
\end{array}
$$

$$
\begin{array}{lcl}
SF & \rightarrow & \#1 \\
\#1 & : & SF.value = SF.inhValue
\end{array}
$$

- Remarque : De nombreuses sémantiques vont suivre le même patron *Visiteur*,
  - Typage : remplacer `value` par `type` et `Compute` par `TypeCheck`
  - Génération de code : remplacer `value` par `code` et `Compute` par `Generate`.

# Construction de l'Arbre Abstrait

- Arbre de dérivation / Arbre syntaxique : Construit automatiquement à partir de la structure de la grammaire
  - Satisfaisant pour les grammaires $LR(k)$
  - Déformé par l'élimination de la récursivité à gauche et la factorisation pour les grammaires $LL(k)$
- Arbre abstrait :
  - Support pour les étapes suivantes d'analyse sémantique
  - Simplification de l'arbre syntaxique (élimination des nœuds inutiles)
  - Réparation des déformations $LL(k)$
- Modèle de donnée pour l'analyse sémantique : méta-modèle
- Exemple des expressions : voir TD GLS Patron Visiteur



# Exemple : Arbre abstrait pour miniML

- Attribut hérité : Aucun
- Attribut synthétisé : `ast` Arbre abstrait pour l'expression
- Action sémantique : Construction de l'arbre
- Grammaire S-attribuée

$$
\begin{array}{lcl}
E & \rightarrow & E + \ T \ \#1 \\
\#1 & : & E.ast = createBinaryExpression(+, E_1.ast, T.ast)
\end{array}
$$

$$
\begin{array}{lcl}
E & \rightarrow & T \ \#1 \\
\#1 & : & E.ast = T.ast
\end{array}
$$

$$
\begin{array}{lcl}
T & \rightarrow & T * F \ \#1 \\
\#1 & : & E.ast = createBinaryExpression(*, T_1.ast, F.ast)
\end{array}
$$

$$
\begin{array}{lcl}
T & \rightarrow & F \ \#1 \\
\#1 & : & T.ast = F.ast
\end{array}
$$

$$
\begin{array}{lcl}
F & \rightarrow & (\ E\ ) \ \#1 \\
\#1 & : & F.ast = E.ast
\end{array}
$$

$$
\begin{array}{lcl}
F & \rightarrow & - \ F \ \#1 \\
\#1 & : & F.ast = createUnaryExpression(-, F_1.ast)
\end{array}
$$

$$
\begin{array}{lcl}
F & \rightarrow & \texttt{entier} \ \#1 \\
\#1 & : & F.ast = createInteger(\texttt{entier}.value)
\end{array}
$$

$$
\begin{array}{lcl}
F & \rightarrow & \texttt{ident} \ \#1 \\
\#1 & : & F.ast = createIdentifierAccess(\texttt{ident}.value)
\end{array}
$$

# Exemple : Arbre abstrait pour miniML

- Élimination de la récursivité à gauche
- Attribut hérité supplémentaire : `inhAst`

$$
\begin{array}{lcl}
E & \rightarrow & T\ \#1\ ST\ \#2 \\
\#1 & : & ST.inhAst = T.ast \\
\#2 & : & E.ast = ST.ast
\end{array}
$$

$$
\begin{array}{lcl}
ST & \rightarrow & +\ T\ \#1\ ST\ \#2 \\
\#1 & : & ST_1.inhAst = createBinaryExpression(+, ST.inhAst, T.ast) \\
\#2 & : & ST.ast = ST_1.ast
\end{array}
$$

$$
\begin{array}{lcl}
ST & \rightarrow & \#1 \\
\#1 & : & ST.ast = ST.inhAst
\end{array}
$$

$$
\begin{array}{lcl}
T & \rightarrow & F\ \#1\ SF\ \#2 \\
\#1 & : & SF.inhAst = F.ast \\
\#2 & : & T.ast = SF.ast
\end{array}
$$

$$
\begin{array}{lcl}
SF & \rightarrow & *\ F\ \#1\ SF\ \#2 \\
\#1 & : & SF_1.inhAst = createBinaryExpression(*, SF.inhAst, F.ast) \\
\#2 & : & SF.ast = SF_1.ast
\end{array}
$$

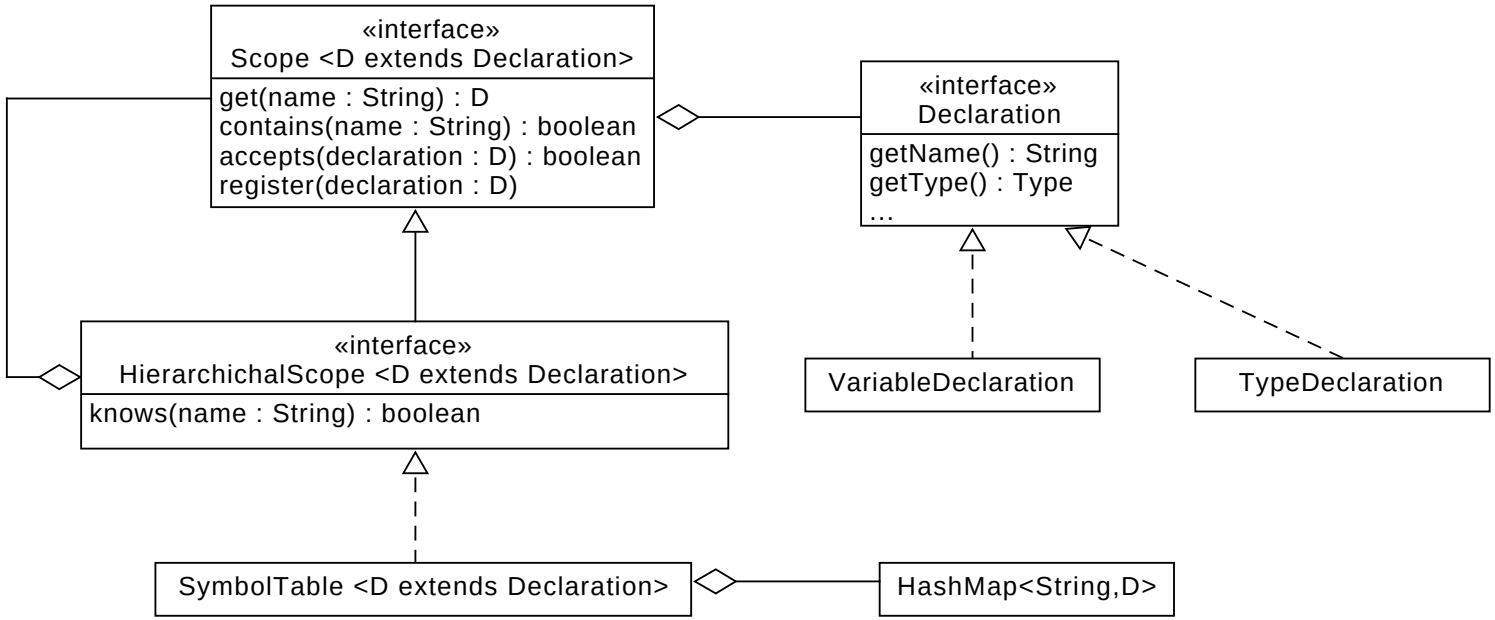
$$
\begin{array}{lcl}
SF & \rightarrow & \#1 \\
\#1 & : & SF.ast = SF.inhAst
\end{array}
$$

# Gestion de la Table des Symboles

- Objectif :
  1. Lier les définitions et les utilisations des identificateurs
  2. Collecter toutes les informations associées aux identificateurs :
     - contenues initialement dans le programme
     - calculées par la sémantique
- Exemple : Environnement d'exécution et de typage de miniML
- Deux approches possibles :
  1. Manipulation explicite d'un dictionnaire (environnement de miniML)
  2. Construction de liens dans l'arbre abstrait
- Gestion de la portée des définitions : Table des symboles hiérarchique
  - Une table pour chaque espace de noms
  - Relations entre tables qui correspondent à l'inclusion (les recouvrements de portée) des espaces de noms

# Architecture de la table des symboles

- Modèle de données fourni :



# Actions sémantiques et Table des symboles

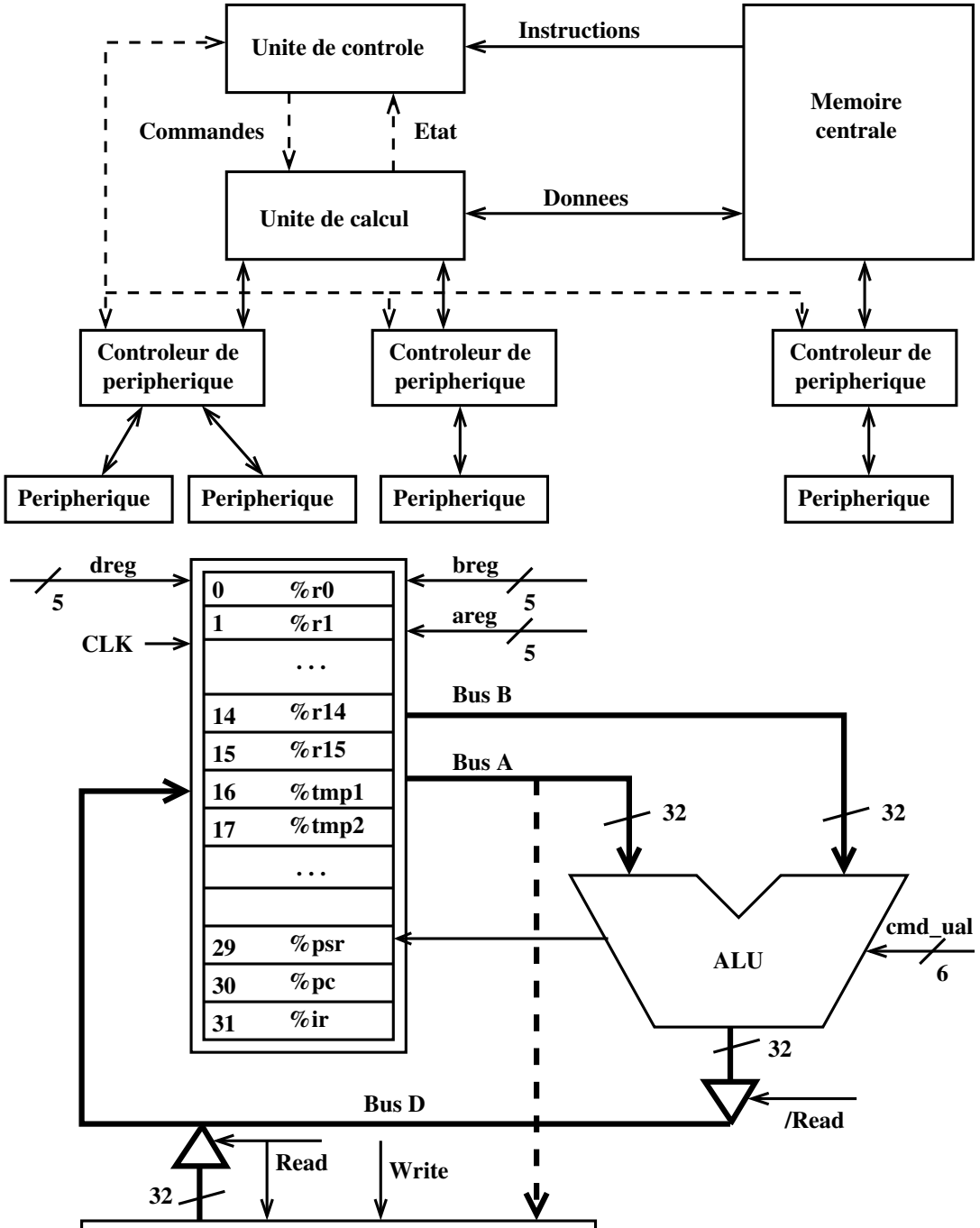
- Actions en relation avec la table des symboles :
  - Collecte des informations
  - Exploitation des informations collectées
- Prise en compte des références en arrière :
  - Collecte et exploitation en une seule étape
- Prise en compte des références en avant :
  - Besoin de collecter l'intégralité des informations avec l'exploitation ;
  - Collecte et exploitation en deux étapes

# Analyse statique et mécanisme de typage

- Rappels :
  - Typage dynamique : Détection des erreurs de typage lors de l'exécution
  - Typage statique : Détection des erreurs de typage avant l'exécution
  - Approche hybride : Si possible avant l'exécution
- Typage fort (strong typing) : Toutes les erreurs de typage à l'exécution sont détectées
- Typage faible (weak typing) : Certains erreurs de typage sont détectées
- Typage souple : En cas de doute, une détection dynamique est ajoutée
- Typage explicite : Tous les identificateurs sont explicitement typés (variables et constantes locales et globales, paramètres et résultats de fonctions, champs d'enregistrements, attributs de classes, etc)
- Typage implicite : Les identificateurs peuvent ne pas être typés et leur type le plus général est calculé par la résolution des contraintes de typage

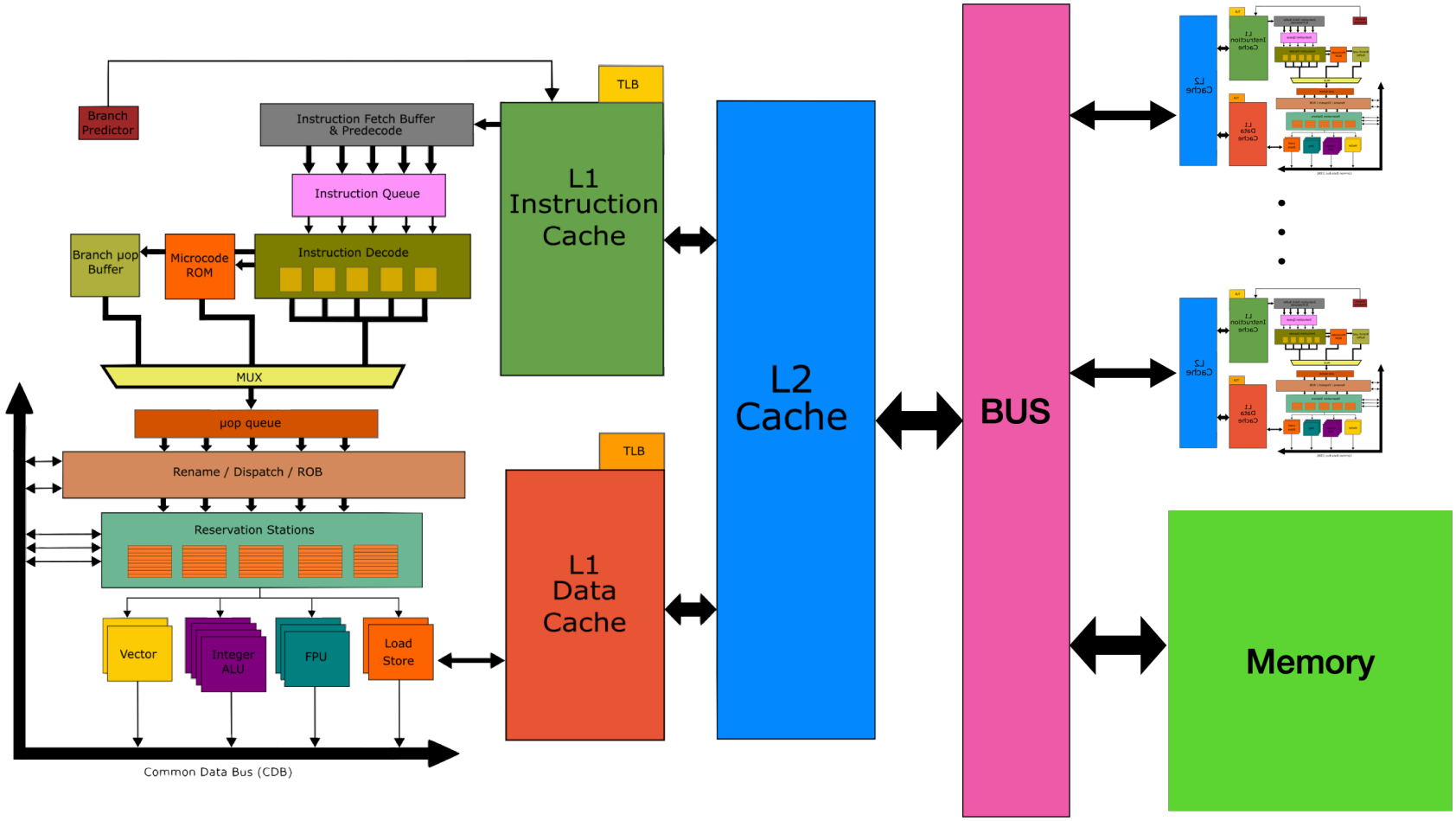
# Rappel : Architecture des processeurs

- Données
  - Mémoire
  - Accumulateurs
  - Registres
- Instructions
  - Unité de contrôle
  - Unité de calcul
  - Contrôleurs de périphérique
- Bus
  - Données
  - Adresses
  - Contrôle



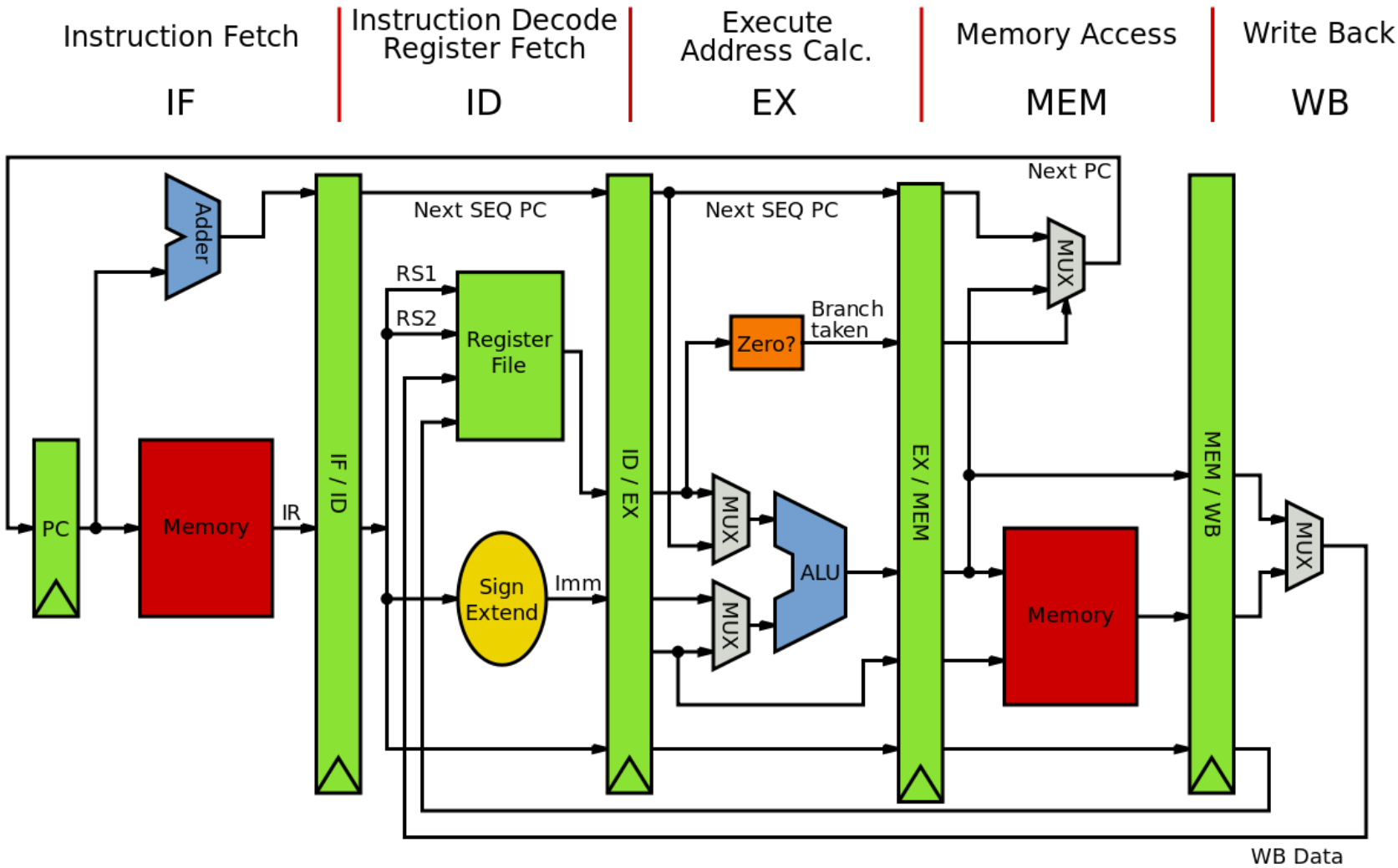
# Complexité des architectures actuelles

- Efficacité de l'accès à la mémoire
- NUMA : Non Uniform Memory Architecture



# Complexité des architectures actuelles

- Parallèlisation du traitement des instructions
- PipeLine, Out of Order, Branch Prediction, Nombreuses unités
- Multi-cœurs, Many-cœurs, Symétriques, Asymétriques



# Placement mémoire

- Objectif : Associer à chaque donnée manipulée un emplacement
- Registre
  - Le plus rapide et économe
  - Nombre limité
  - Problème d'allocation de ressources : Coloration de graphe
  - Parfois imposé par le jeux d'instructions (Instruction Set Architecture)
- Mémoire
  - Pile (Stack) : gérée par le processeur
    - Push, Pop
    - Call, Return
  - Tas (Heap) : géré par le programme
  - Accés :
    - Direct : Adresse constante codée dans l'instruction
    - Indirect (avec ou sans déplacement) : Adresse contenue dans un registre
- Codée dans l'instruction pour constantes
- Codée dans espaces données du programme
- Cache quand possible
- L'emplacement peut dépendre du contexte : Transfert entre mémoire et registres

# Exploitation des machines virtuelles

- Comment gérer l'adaptation à l'architecture des processeurs ?
-
-
- JVM, Java Virtual Machine (Sun puis Oracle)
- CLI/CLR, Common Language Infrastructure/Runtime (MicroSoft)
- LLVM, Low Level Virtual Machine
- ART, Dalvik
- JVM

# La machine TAM

- Dérivée de Triangle Abstract Machine, développée par David Watt pour l'enseignement
- Architecture de Harvard : Instructions et Données séparées
- Mémoire séparée en deux parties contiguës :
  - Pile (Stack) :
    - Adresse croissante
    - gestion par les instructions : `PUSH`, `POP`, `CALL`, `RETURN`
  - Tas (Heap) :
    - Adresse décroissante
    - gestion par la routine : `MAlloc`
- Pas de registres explicitement manipulables
  - SB (Stack Base), ST (Stack Top)
  - HB (Heap Base), HT (Heap Top)
  - LB
  - CB (Code Base), CP (Code Pointer)
- Accès direct :
  - `LOAD`, `STORE` exploitent l'adresse contenue dans l'instruction (registre et déplacement)
  - `CALL` exploite l'adresse associée à l'étiquette
- Accès indirect : `LOADI`, `STOREI`, `CALLI` exploitent l'adresse en sommet de pile

# Gestion des données

- Types de base :
  - Booléen, Caractère, Entier : 1 mot mémoire
  - Manipulés par les routines `SUBR Xxx` qui travaillent sur la pile
- Adresse :
  - 1 mot mémoire
  - Création par instruction `LOADA` et routine `MAlloc`
  - Opérations arithmétiques par les routines sur les entiers
  - Exploitation par accès indirect `LOADI`, `STOREI`, `CALLI`
- Types structurés :
  - Assemblage de types de base : Blocs de taille quelconque
  - Rangement dans la pile ou le tas
  - Stratégie de manipulation :
    - Par composition et décomposition explicite
    - Par calcul de déplacement à la compilation
    - Par calcul de déplacement à l'exécution

# Gestion des données

- Types de base :
  - Booléen, Caractère, Entier : 1 mot mémoire
  - Manipulés par les routines `SUBR Xxx` qui travaillent sur la pile
- Adresse :
  - 1 mot mémoire
  - Création par instruction `LOADA` et routine `MAlloc`
  - Opérations arithmétiques par les routines sur les entiers
  - Exploitation par accès indirect `LOADI`, `STOREI`, `CALLI`
- Types structurés :
  - Assemblage de types de base : Blocs de taille quelconque
  - Rangement dans la pile ou le tas
  - Stratégie de manipulation :
    - Par composition et décomposition explicite
    - Par calcul de déplacement à la compilation
    - Par calcul de déplacement à l'exécution

# Gestion des fonctions

- Enregistrement d'activation :
  - Contient les informations nécessaires à l'exécution de la fonction
  - Paramètres réels
  - Variables locales
  - Adresse de retour
  - Pointeur de pile de retour
  - Lien statique (accès aux variables locales des définitions de fonctions imbriquées)
  - Créé avant l'appel, lors du `CALL` et après l'appel
  - Exploité par le `RETURN`