Synthèse de Languages and Translators
Q8 - LINGI2132

Matthieu Baerts Benoît Baufays Julien Colmonts Alex Vermeylen
Hélène Verhaeghe

Compilation: 15/02/2020 (11:08)

Dernière modification: 02/02/2020 (16:46) 66eb8c8b

Informations importantes Ce document est grandement inspiré de l'excellent cours donné par Pierre Schauss à l'EPL (École Polytechnique de Louvain), faculté de l'UCL (Université Catholique de Louvain). Il est écrit par les auteurs susnommés avec l'aide de tous les autres étudiants, la vôtre est donc la bienvenue. Il y a toujours moyen de l'améliorer, surtout si le cours change car la synthèse doit alors être mise à jour en conséquence. On peut retrouver le code source et un lien vers la dernière version du pdf à l'adresse suivante

https://github.com/Gp2mv3/Syntheses.

On y trouve aussi le contenu du `README` qui contient de plus amples informations, vous êtes invités à le lire.

Il y est indiqué que les questions, signalements d'erreurs, suggestions d'améliorations ou quelque discussion que ce soit relative au projet sont à spécifier de préférence à l'adresse suivante

https://github.com/Gp2mv3/Syntheses/issues.

Ça permet à tout le monde de les voir, les commenter et agir en conséquence. Vous êtes d'ailleurs invités à participer aux discussions.

Vous trouverez aussi des informations dans le wiki

https://github.com/Gp2mv3/Syntheses/wiki

comme le statut des documents pour chaque cours

https://github.com/Gp2mv3/Syntheses/wiki/Status

Vous pouvez d'ailleurs remarquer qu'il en manque encore beaucoup, votre aide est la bienvenue.

Pour contribuer au bug tracker et au wiki, il vous suffira de créer un compte sur GitHub. Pour interagir avec le code des documents, il vous faudra installer LaTeX. Pour interagir directement avec le code sur GitHub, vous devrez utiliser `git`. Si cela pose problème, nous sommes évidemment ouverts à des contributeurs envoyant leurs changements par mail (à l'adresse contact.epldrive@gmail.com) ou par n'importe quel autre moyen.

Table des matières

Todo list

ben

CFG : Context-Free Grammar

PEG : Parsing Expression Grammar : grammaire non-ambigue qui est exprimmée comme une CFG mais pour éviter les ambiguités, on prend tjs la première dérivation qui marche.

1 Chapitre 1 : Compilation

1.1 Compiler

Compilateur Programme qui traduit un programme source écrit dans un langage haut niveau en un programme équivalent écrit en langage de bas niveau (code machine le plus souvent) qui peut être exécuté directement par l'ordinateur.

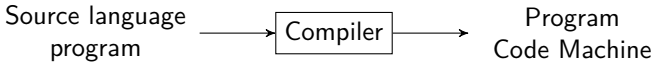


FIGURE 1 – Compilation

1.1.1 Programming language

Langage de programmation Un langage artificiel dans lequel un programmeur écrit un programme pour contrôler le comportement d'une machine.

On définit un langage de programmation en 3 étapes :

— les **tokens** : « mot-clé » du langage

— la **syntaxe** et la **construction** du langage : comment écrire un programme qui fonctionne

— la **sémantique** du langage : sens des phrases.

1.1.2 Machine language

Langage machine (ou ensemble d'instructions) est un langage facilement interpreté par l'ordinateur (le processeur plus précisement) lui-même. Chacune des instruction occupe un byte ou plus et est facilement accédé et interprété. On parle également d'architecture pour l'ensemble d'instructions et le comportement d'une machine.

Complexité instruction On distingue plusieurs niveaux de complexités (CISC : *Complex Instruction Set Computer* ou RISC : *Reduced Instructive Set Computer*). Il faut plusieurs instructions RISC pour obtenir une instruction CISC.

Comme les registres sont très rapides, un ordinateur va essayer de garder au maximum les données dans les registres.

Machine virtuelle L'architecture de la machine est implémentée en software et cette implémentation exécute un programme comme une machine physique. Exemple d'une machine virtuelle : la JVM (java virtual machine). L'avantage d'une machine virtuelle est que les programmes générés sont indépendents de l'architecture réelle de la machine, il suffit qu'il existe une machine virtuelle pour cette architecture.

Les étapes de l'analyse du compilateur pour produire le programme de sortie sont :

— Mapper les noms aux adresses mémoires, frames de la pile et registres.

— Générer la liste linéaire des instructions machines.

— détecter les erreurs

1.1.3 Interpreteur Vs Compilateur

Dans le cas d'un interpréteur, le code high-level est exécuté directement.

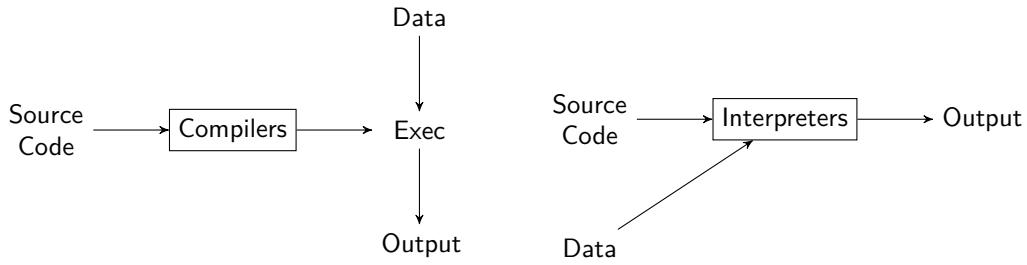


FIGURE 2 – Compilation

Avantages compilateur	Avantage interpréteur
— **Performances** : le code machine est plus rapidement exécuter, l'interpréteur doit lui redécoder à chaque fois les lignes de codes pour les transformer en instructions. — **Secret** : le code source est très difficile à déduire depuis un code machine.	— L'overhead de l'interpreteur ne nécessite pas toujours la mise en place d'un compilateur (ex : les scripts) — **Contrôle** : le code source est disponible, facilement modifiable. — **Portabilité** : le langage machine dépend de la machine alors qu'un interpréteur est portable.

1.2 Why should we study compilers ?

— Ce sont de très grands programmes (comme ceux que nous étudierons plus tard)
— Il fait usage de tout ce que nous avons vu avant
— Cela permet d'apprendre sur le langage
— On en apprend sur la machine visée
— Il existe encore du boulot de compilation pour les nouveaux langages
— Ils se retrouvent partout
— Les programmes utilisent du XML se servent de technologies de compilation
— Mix de théorie et pratique
— Comme un compilateur peut être écrit en étapes, c'est un cas d'étude d'ingénérie logicielle
— écrire des programmes est chouette

1.3 How does a compiler work ? Phases of compilation

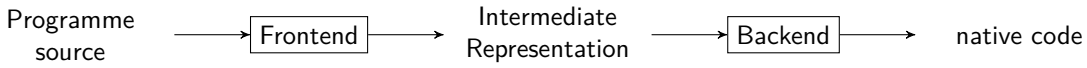


FIGURE 3 – Compiler work

1.3.1 Frontend

— Analyse du programme pour en deviner le sens
— dépend uniquement du langage de départ

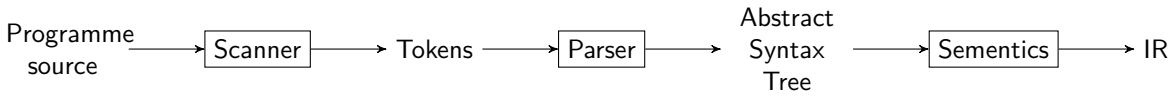


FIGURE 4 – Frontend

1.3.2 Backend

— Prend le IR et produit le code machine

— Dépend de la machine d'arrivée uniquement

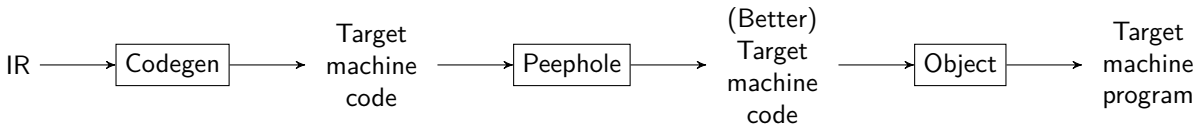


FIGURE 5 – Backend :

— Codegen : Choisi les instructions adéquates

— Peephole : Cherche à faire un peu d'optimisation locale du code

— Object : liste les différents modules et construit un programme exécutable

1.3.3 Middle end

Il peut y avoir l'ajout d'un optimisateur entre le frontend et le backend. Celui-ci à pour but d'améliorer le code intermédiare :

— Il organise le programme en blocks

— Recherche la durée de vie des variables (next-use information)

— Élimine des sous-expressions et des *constaints folding* ($x+4=9$ si on sait que $x=5$), regarde à l'allocation des registres

— Met les invariants hors des boucles, remplace les multiplication par des additions (car moins couteuse)

1.3.4 Avantage de la décomposition

— Réduit la complexité

— Permet un dévelopement en parallèle

— Permet une réutilisation (un seul backend pour tous les langages pour une même machine)

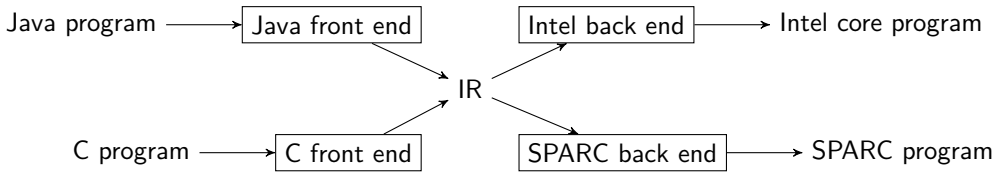


FIGURE 6 – Decomposition

1.3.5 Compiling to a Virtual Machine : New Boundaries

.class Les programmes Java sont d'abord compilés en un fichier `.class` (a **byte-code**) éxécutable sur la machine virtuelle. Le `.class` peut donc être considérer comme un **IR** (Intermediate Representation), le `JavaC` comme le frontend et la machine virtuel comme le backend.

Machine virtuel La machine virtuelle est un interpréteur qui observe le code qu'on lui donne et compile les parties critiques (*hotspot*) qui sont les plus souvent appelées. Il fait aussi du *in-lining*, c'est-à-dire remplacer des appels de méthode par leur code.

Note : Sur Windaube, ils ont implémenté une machine CLR (Commun Langage Runtime). Suivant une technique appelée JIT (Just In Time), le CLR compile chaque méthode en code natif et utilise ce code lorsque la méthode est appelée.

1.3.6 Compiling JVM code sur une architecture en registre

En effet, le JVM code à une architecture basé sur la stack et il peut y avoir différent challenge à résoudre pour mapper beacoup de variables vers un nombre limité de registre rapide.

La stratégie consistant à fournir du code intermédiare pour une machine virtuelle a plusieurs avantages :

— Le code intermédiare est compact

— De gros efforts ont eété fait pour optimiser ces machines virtuelles pour qu'elles s'exécute très rapidement (ex : Java, LOL)

— Le fait de ne compiler que les régions spéciales permettent d'aller plus vite que de compiler l'entièreté du code.

1.4 An overview of the j-- to SVM compilers

j-- est un sous-ensemble de Java : non-trivial, orienté objet, supportant les classes, méthodes, champs, messages, statements, expressions et types primitifs.

Son compilateur est écrit en Java d'une manière orienté objet.

1.4.1 j-- compiler organization

Le début est la `main`. Après avoir lu les arguments, elle crée le parser et le scanner. Ensuite :

— `compilationUnit()` au parser pour obtenir l'AST.

— `preAnalyze()` au nœud principal de l'arbre pour déclarer les types et classes dans la table des symboles.

— `analyze()` pour déclarer les noms et chercher les types

— `codegen()` pour gérer le code

— S'il y a des erreurs, on en cherche d'autres et puis on arrête.

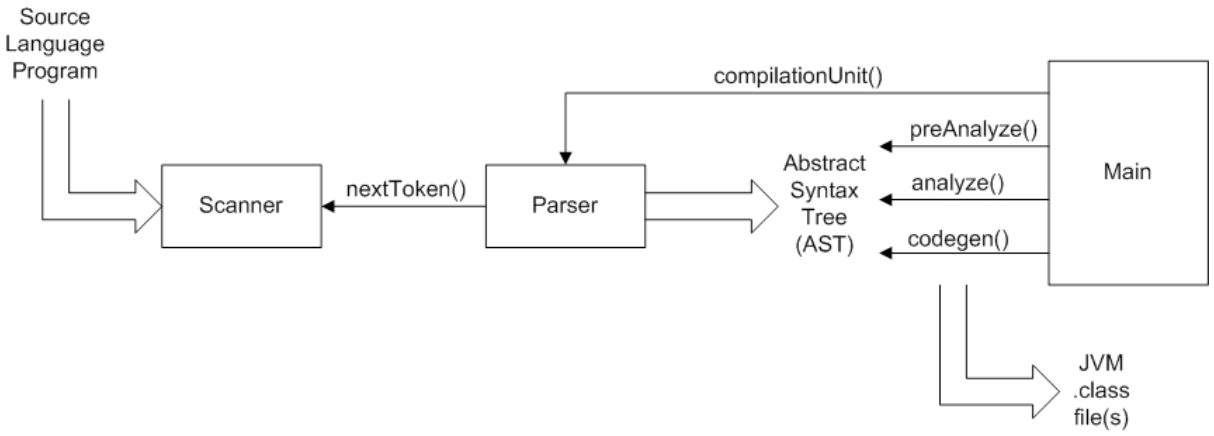


FIGURE 7 – j-- compiler

1.4.2 Scanner

Le scanner scanne les tokens à la demande et les classes en :

- **Identifiant** (`main, myMethod`)
- **tokens réservés** (`import, public, class, operateurs, separateurs,...`)
- **litéral** (tout le reste sans les séparateurs : `"Hello World", 3`, etc.).

1.4.3 Parser

Le parser est spécifique à la syntaxe du langage et est effectué en *recursive descent*.

1.4.4 AST

L'abstract syntax tree est une représentation sous forme d'arbre du programme source. Il permet de rendre explicite la structure syntaxique. Chacune des classes des nœuds étendent une même classe (`JAST`) et implémente 3 méthodes :

- `preAnalyze()` (déclarer les types)
- `analyze()` (déclarer les variables locales)
- `codegen()` (pour générer le code).

1.4.5 Types

Un type indique comment une varaible doit être interpreté et se comporte. Java est dit statiquement typé car les types sont déterminés à la compilation.

Dans le compilateur j--, on crée une classe type pour placer les différents types.

1.4.6 Symbol table

1.4.7 `preAnalyze()` and `analyze()`

`preAnalyze()` l'étape de pré analyse doit traverse l'AST mais seleument jusqu'à un certain point pour :

- declarer les types importés
- declarer les classe définie pas le programmeur
- declarer les attributs des classes
- declarer la signature des méthodes (nom et type des paramètres)

Il construit une partie de la table des symbol qui est au dessus du AST.

Cette étape est nécessaire car dans le code source on peut référencer certains types avant de les avoir déclarés.

`analyze()` reprend là où `preAnalyze()` s'est arrêté. Il s'occupe de :

- Type checking : trouver le type de chaque expression
- Accessibilité : Regarde aux `public`, `protected` et `private`
- Member finding : vérifie les signatures.
- Tree rewriting : réécrit certains tree

1.4.8 Stack Frames

L'analise s'occupe aussi de calculer l'offset des variables locales dans la stack frame (block continu de mémoire au-dessus de la pile *run-time*). On peut ainsi savoir l'espace repris par une méthode à chacune de ses invocations.

1.4.9 `codegen()`

Permet de générer le byte code pour la JVM. Un outil (`CLEmitter`) existe et permet de construire la table des constantes et de générer les références utilisées par la JVM pour les noms et constantes. Permet de calculer des offsets de *brckets* et les adresses. Calcule les coûts (espace mémoire) des variables et l'espace de calcul nécessaire dans une méthode. Et pour finir, il construit le fichier `.class`.

1.4.10 j-- compiler source tree

Pour ajouter une opération, on doit modifier le scanner pour reconnaitre les nouveaux tokens, modifier le parser pour reconnaitre l'expression, implémenter l'analyse sémentique et finalement la génération du code allant avec cette opération.

2 Chapitre 2 : Lexical Analysis

2.1 Introduction

FIGURE 8 – Lexical analysis

L'analyse lexical parse donc l'input en une série de **token**.

Lexical tokens : éléments composant les programmes. Ils sont décrits par la syntaxe lexicale du langage.

Lexical analysis : procédé permettant d'identifier les lexical tokens dans le programme. On peut séparer les tokens lexicaux en catégories :

- les identifiants
- les mots réservés, opérateurs et séparateurs
- les literaux

2.2 Scanning tokens

Scanner Un scanner est un programme écrit par le programmeur (*handwritten*) ou bien généré automatiquement par une liste de regex comme JavaCC.

Les espaces peuvent servir à séparer des tokens mais pas tout le temps, par exemple si on est entrain de scanner un `integer` et que l'on tombe sur une lettre, on peut en conclure que l'`integer` est fini.

2.3 Lexical analysis hand written

Dans le cas d'un scanner écrit à la main, il est très utile de décrire le scanner comme un **state transition diagram**.

2.3.1 State transition diagram

diagramme dans lequel les nœuds représentent les états, les flèches dirigées représentes des mouvement d'un état à l'autre suivant ce qui est scanné. Si un caractère scanné n'est sur aucune flèche, on choisi celle non étiquetée.

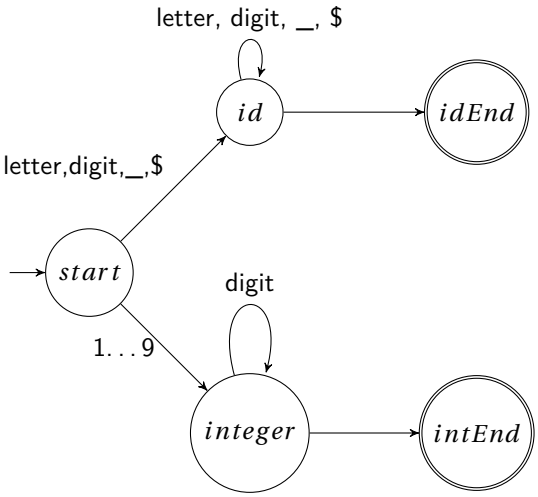


FIGURE 9 – State diagram

Implémentation

- Les choix sont représenté par des `if` (ou `switch`)
- Les cyles sont des `while`
- Les mots réservés ne sont pas intégré dans la machine à états car cela est trop complexe. A la place on liste les mots réservés dans une table et faire un *lookup* à chaque fois qu'on reconnait un identifiant.
- Les opérateur sont intégré dans la machine à états finis (via un `switch` entre les différents opérateurs possible)
- Les espaces sont ignorés (indique la fin d'un token)
- Les commentaires sont ignorés

2.4 Lexical analysis generator

FIGURE 10 – Generator lexical analysis

2.4.1 Regular Expressions (regex)

On dit que les regex définissent un langage de strings sur un alphabet.

2.4.2 Finite State Automate (FSA)

Pour tout langage décrit par une regex, il existe un diagramme à transition d'état.
Un automate fini reconnait un language.

Automate à états finis (Finite State Automate) c'est $F = (\Sigma, S, s_0, M, F)$ où :

- Σ est l'alphabet
- S est l'ensemble des états
- $s_0 \in S$ est l'état initial
- M est l'ensemble de mouvements possibles d'un état à un autre ; $M(r,a) = S$ où $r, s \in S$ et $a \in \Sigma$
- $F \in S$ est l'ensemble des états finaux

Input	Language	regex	contient
$a \in$ alphabet	$L(a)$	a	Le string a
r et s des regex	$L(rs)$	rs	Toute les combinaisons de concaténation rs
r et s des regex	$L(r\|s)$	$r\|s$	$L(r) \cup L(s)$
r une regex	$L(r*)$	$r*$	Concaténation de zéro ou plusieurs instance de r

— ϵ est le langage contenant uniquement la string vide

— r et (r) sont les mémes regex. Les parenthèses sont utilisées pour grouper les regex.

TABLE 1 – Formes possibles des RegEx

Une phrase appartient au language si en partant du start, on arrive sur un état final après avoir suivit les changement d'états correspondant à la phrase.

Il existe un moyen automatique de générer un FSA à partir de regex.

Exo : Spécifiez un automate à état fini pour (ab)*ab

2.4.3 Non-Deterministic Finite State Automate (NFA) vs. Deterministic FSA (DFA)

DFA automate à états finis où il n'y a pas de ϵ-move et où un seul chemin existe à partir d'un état pour un input.

On ne peut donc aps avoir $M(r,a) = s$ et $M(r,a) = t$ tel que $s \neq t$.

— DFA are faster to execute and easy to implement (by table driven implementation)

NFA automate à états finis qui permet plusieurs chemins à partir d'un état pour le même input, et autorise ϵ-move.

— NFA are in general smaller (exponentially)

2.4.4 Regular expressions to NFA

On utilise les constructions de **Thompson**.

2.4.5 NFA to DFA

Il est évident qu'un NFA nécessite du **backtracking** poour parser un input (puisqu'il peut emprunter le mauvais choix) qui prend du temps. . . Heureusment, pour toutes NFA il existe une DFA équivalente. On arrive à la trouver grâce aux ϵ-closure.

ϵ-closure(s) c'est un ensemble incluant s ainsi que tous les états atteignables à partir de s uniquement avec des ϵ-moves.

ϵ-closure(S) c'est un ensemble incluant l'ensemble S ainsi que tous les états atteignables à partir de chaque élément s de S uniquement avec des ϵ-moves.

2.4.6 Minimal DFA

L'idée est de fusionner des états non-distinguable (càd qu'on ne peut distinguer d'un autre pour chaque input). On va recréer des états grace à des partitions.

```
Input: a set of states, S.
Output: ϵ-closure(S)

Stack P.addAll(S);
Set C.addAll(S);
while (!P .empty() ) {
    s = P.pop();
    for (r ∈ move(s, ϵ)){
        if (r ∉ C) {
            P.push( r );
            C.add( r );
        }
    }
}
return C;
```

FIGURE 11 – ϵ-closure algorithm

```
Input: an NFA, N = (∑, S, s0 , M, F)
Output: DFA, D = (∑, SD , sD0 , MD , FD )

Set SD0 = ϵ-closure(s0 );
Set SD .add(SD0 );
Moves MD ;
Stack stk.push(SD0 );
i = 0;
while (! stk.empty()) {
    t = stk.pop();
    for (a : ∑) {
        SDi+1 = ϵ-closure(M(t, a));
        if (SDi+1 != {}) {
            if (SDi+1 ∉ SD ) {
                // We have a new state.
                SD .add(SDi+1 );
                stk.push(SDi+1 );
                i = i+ 1;
                MD .add(MD(t, a) = i ) ;
            }
            else if (∃ j, Sj ∈ SD ∧ SDi+1 == Sj ) {
                .. In the case that the state already exists.
                MD .add(MD(t,a) = j ) ;
            }
        }
    }
}

Set FD ;
for (sD : SD )
    for (s : sD )
        if (s ∈ F)
            FD .add(sD );

return D = (∑, SD , sD0 , MD , FD );
```

FIGURE 12 – DFA construction algorithm

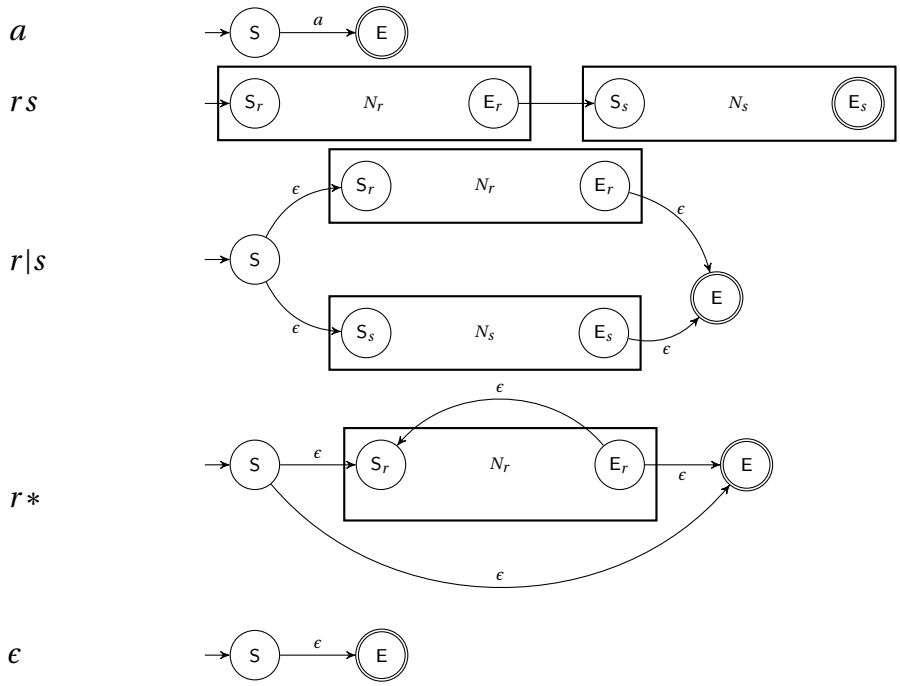


TABLE 2 – Construction de thompson

Partitions

— Les partitions initialles sont *final* et *non-final*.

— On scinde en différente partition quand les états sont distinguables

— Pour chaque partition, un input doit induire un mouvement vers une même partition.

Note : On doit s'assurer que pour chacun des états, le mouvement d'un symbole particulier reflète celui dans l'ancien DFA.

2.4.7 Table driven

For each DFA, a Table driven can be implemented as a 2D table (states and input).

```
For every M(r,a) = s, T[r,a] = s.
```

Recognize the longest match Un automate reconnait les états finaux avec deux variables :

— `Last-Final` : le dernier état final rencontré

— `Input-Position-at-Last-Final`

On met à jour ces deux variables quand un état final est rencontré. Quand a *dead* état (càd un état qui n'a pas l'ouput correspondant à l'input) est atteint, les variables disent quels tokens ont matché et où ils se sont fini.

2.4.8 JavaCC : Tool for generating scanner

JavaCC est un outil pour générer une analyse lexicale à partir de regex ainsi qu'un parseur à partir d'une grammaire hors contexte.

Lorsque l'on scanne un token, on considère tous les regex correspondant à l'état actuel et on choisi celui qui contient le plus de caractères.

Il existe 4 types de regex :

— SKIP : regex jeté

— MORE : on continue après

— TOKEN : crée un token et est retourné au parseur

— SPECIAL-TOKEN : crée un token spécial ne participant pas au parsing

3 Chapitre 3 : Parsing

3.1 Introduction

Parsing action de mettre les tokens ensemble pour créer des entités syntaxiques plus grandes (expressions, instructions, méthodes, définition de class, ...).

Fonction du parsing :

— Assumer la validité syntaxique du programme.

1. Identifier une erreur et la reporter (erreur et le numéro de ligne).
2. Ne pas s'arrêter après une erreur et chercher d'autres erreurs (pour ne pas reporter qu'une seule erreur à la fois).

— Fournir une représentation du programme parsé, souvent en arbre (**AST**) car ce type de représentation est facile à analyser et à "décorer" d'informations.

3.2 Context-Free Grammars and Languages

CFG est plus puissant que RegExp. En effet, impossible en RegExp de décrire un string tel que une suite de “a” est suivit d'une suite de “b” tel que les deux séquences ont la même taille.

Cela est impossible puisque les DFA ne peuvent pas compter !

```
S := a S b
S := ϵ
```

3.2.1 Backus-Naur Form (BNF) and its extensions

On définit les langages de programmation de manière récursive.

::= : "*peut être écrit comme*", sigle de la définition

| : "ou"

$[X]$: X est optionnel

$\{X\}$: *Keene closure*, X apparait de 0 à plusieurs fois.

3.2.2 Grammar and the Language it describes

Context-Free grammar Une grammaire sans contexte est un tuple $G = (N, T, S, P)$ où :

N est un ensemble de symboles non terminaux

T est un ensemble de symboles terminaux

$S \in N$ est un non terminal appelé symbole de départ

P est un ensemble de règles de production

Derivation Séquence d'applications de règles de production, démarrant du symbole de départ pour arriver à l'expression voulue (« $\Rightarrow$ » est le symbole de la relation de dérivation).

Directly derive Lorsqu'un string de symboles dérive d'un autre en une seule étape.

E directly derives T + T : $E \Rightarrow T + T$

Language $L(G)$ On dit que le langage $L(G)$, décrit par une grammaire G, consiste en tous les strings ne comprenant que des symboles terminaux pouvant être dérivés du symbole de départ.

Left-most derivation Dérivation dans laquelle, à chaque étape, le string suivant est dérivé en applicant une règle de production réécrivant le non terminal le plus à gauche.

Right-most derivation Dérivation dans laquelle, à chaque étape, le string suivant est dérivé en applicant une règle de production réécrivant le non terminal le plus à droite.

Sentential form Chaque string de terminaux ou non terminaux qui peut être dérivé à partir d'un symbole de départ.

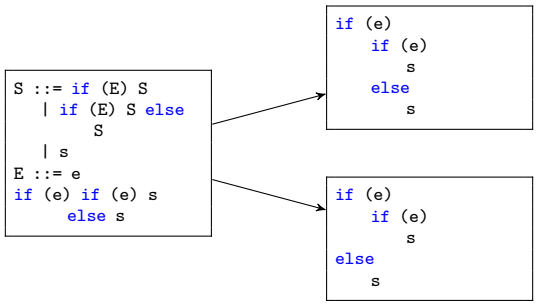
Parse tree Arbre illustrant les dérivations effectuées depuis un symbole de départ (racine) pour arriver à une expression (feuilles).

3.2.3 Ambiguous Grammars and unambiguous grammars

Detecter si une context free grammar est ambiguous est **indécidable**. Notons que les grammaires LL(k) et LR(k) sont non-ambiguous.

- **Ambiguous sentence** : Une phrase par laquelle il existe plusieurs dérivations possibles. (2 arbres pour représenter $id + id * id$)
- **Ambiguous grammar** : Une grammaire qui décrit au-moins une phrase ambigüe.
- **Unambiguous grammar** : Une grammaire ne décrivant que des phrases à dérivation unique (qu'elle soit *left* ou *right*).

IF..ELSE ambiguous Pour résoudre cet ambiguité, on définit une règle telle que le `else` se groupe avec le `if` précédent.



x.y.z ambiguous Il y a aussi ambiguité avec $x.y.z$: est-ce un *field selection*, un nom de package, un nom de class, etc.

Pour résoudre cela, un nœud `Ambiguous Name` est placé dans l'**AST** et est revu par le compileur après la déclation des types (*type declaration*).

3.2.4 Parseur

Pour les parseur des langages décrits par une grammaire *context-free*, on utilise une *pushdown* stack avec *backtracking*.

Lorsque le *backtracking* n'est pas nécessaire, on parle de parsing déterministe. Deux types de parsing déterministes :

- Top-Down : le parser commence avec le `start symbol` et étape par étape dérive l'input sentence pour produire l'**AST** du root aux feuilles.
- Bottom-up : le parser commence avec l'input sentence et le scanne de gauche à droite et applique les règles. L'**AST** est construit du bas vers le root.

3.3 Top-Down Deterministing Parsing

Deux types de top-down :

— recursive descent

— $LL(1)$.

Le parser commence au symbol de départ (root) et agrandit l'AST en déscendant vers les feuilles... il remplace le symbole par le coté droite d'une règle BNF.

Il va scanner la phrase d'input et parser les entités syntaxique représentées par le symbole de départ.

3.3.1 Parsing by recursive descent

Parser par descente récursive invoque une procédure écrite pour parser chaque non-terminal selon qui lui est associé. La procédure choisit la règle a appliquer selon le prochain non-scanné symbol. (Ne nécessite pas de backtracking !)

Ces méthodes construisent également les nœuds AST !

Utils Le scanner a quelques méthodes utiles pour aider à décider de la règle de production adéquate :

— `have(TOKEN)` : renvoie `true` si le token est présent, et le scanne.

— `mustBe(TOKEN)` : renvoie `true` si le token est présent et le scanne ou une erreur sinon.

— `see(TOKEN)` : regarde si le token est présent dans le scanner.

Lookahead Pour certaines grammaires, il faut regarder plus loin que le token suivant pour savoir ce qui est présent en premier et pour savoir quelle loi de dérivation utiliser. On a un *Lookahead* scanner définit qui permet de voir plus loin qu'un symbole.

Error recovery Pour récupérer des erreurs, les `mustBe`, lorsqu'il rencontre un token inatendu, scanne jusqu'à trouver celui qu'il veut jusqu'à se remettre dans un état à nouveau correct.

3.3.2 Left recursion removal

Parfois pour faire un parsing de descente récursive, il faut modifier la grammaire même si elle n'est pas ambigüe. Par exemple pour la grammaire :

— Y : := Y a

— Y : : = b

on ne peut pas faire le parseur à cause de la left-recursion. On a de la left-recursion quand le symbole non-terminal le plus à gauche de la règle de production et le méme que le déclencheur. Il faut retirer cette recursion en faisant :

— Y : := b Y'

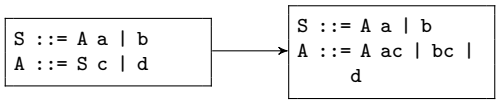
— Y' : := aY'

— Y' : :=ϵ

L'algo qui permet de le faire est le suivant :

```
Enumerate the non-terminal of G: X1,X2,..,Xn
for i := 1 to n do
       for j:= 1 to i-1 do
              Replace each rule in P having the form
              Xi::= Xjα by the rules
                     Xi::= β1α|β2α|...|βkα where Xj ::= β1|β2|..|βk
                     Eliminate any immediate left recursion
```

Indirect left recursion Dans ce cas la, une première manipulation est nécessaire avant d'appliquer la régle de base.



Left Factoring Left Factoring consiste à retirer les facteurs gauche communs dans des règles de productions du même symbole terminal. Par exemple

— Y : := $\alpha\beta$

— Y : := $\alpha\gamma$

deviens

— Y : := α Y'

— Y' : := $\beta|\gamma$

Cette technique est utile parce qu'elle permet d'éviter le backtracking (donc utile pour les lookahead scanner)

3.3.3 $LL(1)$ parsing

Contrairement au recursive descent, ici, la stack est explicite. Le 1^{er}L veut dire *left-to-right scan*, le 2^{e}pour *left-most derivative* et le 1 pour le fait qu'on regarde un seul symbole à la fois.

Chaque grammaire a une table de parsing unique. Pour créer la table, si α et β sont des strings (pouvant être vides) de terminaux et non terminaux :

$$table[Y,a] = i \text{ où } i \text{ est le numéro de la règle, } Y ::= X_1, X_2, ..., X_n \text{ si soit } X_1, X_2, ..., X_n \Rightarrow a\alpha \text{ ou } X_1, X_2, ..., X_n \overset{*}{\rightarrow} \epsilon \text{ et il y a une dérivation } S\# \Rightarrow \alpha Ya\beta.$$

Pour arriver à établir cette table, on fait usage de deux ensemble : `first` et `follow`.

LL(1) parsing algorithm Au tout début, on place le symbole terminal initial sur la pile. Suivant le token scanné, on remplace le symbole sur la pile par celui qui correspond suivant la table de parsing.

Lorsqu'un non-terminal est trouvé sur la pile et si c'est bien celui qui est scanné, il est retiré de la pile et un nouveau token est scanné.

```
Initially,
Stack stk initially contains the terminator # and the start symbol S, with S on top;
Symbol sym points to the first symbol in the sentence w.
for (;;) {
    Symbol top = stk.pop();
    if (top == sym == #) halt successfully;
    else if (top is a terminal symbol) {
        if (top == sym) {
            advance sym to point to the next symbol in w;
        } else {
            halt with error: a sym found where a top was expected;
        }
    } else if (top is a non-terminal, Y) {
        int index = table[Y,sym];
        if (index != err)} {
        rule = rules[index];
        Say rule is Y ::= X 1 X 2 ...X n
        Push X n,..., X 2, X 1 onto the stack stk, with X 1 on top
        }
        else {
            halt with an error.
        }
    }
}
```

First Correspond à l'ensemble des terminaux qui peuvent commencer une règle. **First may contain ϵ ($X_1X_2...X_n \Rightarrow *\epsilon$)**

$$first(X_1X_2...X_n) = \{a \quad | \quad X_1X_2...X_n \Rightarrow *a\alpha, \quad a \in T\}$$

1. Calcul de $first(X)$ pour tous les X dans la grammaire :

```
For each terminal x, first(x) = {x}.
For each non-terminal X, first(X) = {}.
If there is a rule X ::= ε in P
    first(X) += ε

Repeat until no new symbols are added to any set:
    For each rule Y ::= X1X2...Xn ∈ P
        first(Y) += all symbols from first(X1X2...Xn)
```

2. Calcul de $first(X_1X_2...X_n)$:

```
Set S = first(X1).
i = 2
While ε ∈ S and i ≤ n {
    S = S - ε
    S += first(Xi)
    i = i+1
return S
```

Follow Correspond à l'ensemble des terminaux qui peuvent suivre une règle. **Follow cannot contain ϵ**

$$follow(X) = \{a \quad | \quad S \Rightarrow *wX\alpha \quad and \quad \alpha \Rightarrow *a...\}$$

Calcul de $follow(X)$:

```
follow(S) = {\#}.
follow(X) = {} for all non-terminals X ≠ S.
Repeat until no new symbols are added to any set:
    for each rule Y ::= X1X2...Xn ∈ P,
        for each non-terminal Xi,
            follow(Xi) += first(Xi+1Xi+2...Xn) - ε
            if Xi is the rightmost symbol OR first(Xi+1Xi+2...Xn) contains ε,
                follow(Xi) += follow(Y)
```

Parsing table Construire une table de parsing $LL(1)$ pour une grammaire $G = (N, T, S, P)$:

```
For each non-terminal Y in G,
    For each rule Y ::= X1X2...Xn ∈ P with number i
        For each terminal a ∈ first(X1X2...Xn) - {ε}
            table[Y,a] += i

        If first(X1X2...Xn) contains ε
            for each terminal (or \#) in follow(Y)
                table[Y,a] += i
```

Une grammaire est dite $LL(1)$ si la table de parsing produite n'a pas de conflits. Toute grammaire $LL(1)$ est non ambigue.
On peut avoir des grammaire non $LL(1)$ mais $LL(k)$ avec $k > 1$ (on regarde alors k symboles).

Error recovery Si il reçoit un input non attendu pour terminal/non-terminal (T) dans la stack, il va simplement skipper les tokens jusqu'a obtenir un token $\in$ Follow(T).

3.4 Bottom-up deterministic parsing

On passe ! !

3.5 Parser Generation using Java CC

On utilise une structure syntaxique EBNF (*extended BNF*). On peut définir ainsi des parseurs $LL(k)$ récursive descent. Le fichier `j-.jj` contient donc les regex pour la structure lexicale ainsi que les règles syntaxiques du langage.

Entre `PARSER_BEGIN(JavaCCParser)` et `PARSER_END(JavaCCParser)` sont définies des fonctions utiles et utilisées dans le parser généré : `reportParserError()` et `recoverFromError()`.

3.5.1 Specification syntaxique

La syntaxe EBNF est :

— $[a]$: 0 ou 1

— $(a)^*$: 0 ou plusieurs

— $a|b$: a ou b

— () : grouping

3.5.2 Déclaration de symbol non-terminaux

On définit premièrement un symbole de départ (`compilationUnit`) qui est un "haut niveau" de non-terminal et référence le niveau inférieur de non-terminal. Ceux ci reference les tokens définit dans la spécification lexical.

La déclaration de non terminaux ressemble à une méthode java. Il y a un type de retour, un nom, peut avoir des arguments et à un corps de méthode. Il y a un bloc précédent celui-là dans lequel les variables locales sont déclarées.

3.5.3 Lookahead

Lorsqu'il est nécessaire de regarder plusieurs symboles après l'actuel, on utilise la méthode `LOOKAHEAD(<token>)`.

Il y a deux méthodes pour récupérer des erreurs : *shallow* et *deep* (utilisé par j--). Dans le corps d'un non terminal, on catch les erreurs du parseur. On choisit alors à quel token on veut reprendre (*skip to token*). Il y a donc premièrement affichage d'une erreur puis récupération par shipping.

3.5.4 Avantage

Les avantages d'un parser autogénéré (JavaCC) sont :

— Structure lexicale mieux expliquée

— Construction EBNF possible

— Lookahead possible et facile

— Conflit de choix reporté

— Mécanisme de récupération d'erreur sophistiqué utilisable.

3.6 Bactracking parsing (not in the book)

Avec le backtracking parsing, on peut parser n'importe quel langage décrit par une CFG. On va dériver en left most à partir du symbol initial en choisisant la première règle de dérivation. Tant que les terminaux déjà obtenu match le string qu'on veut matcher alors on continue, sinon on revient à l'étape pérédente et on choisit la règle de dérivation suivante.

Exemple :

Suivant la grammaire :

```
S ::= ASA
S ::= ASB
S ::= A
A ::= a
B ::= b
```

Si on souhaite parser aab, on a les étapes suivantes :

```
S
S => ASA
S => ASA => aSA
S => ASA => aSA => aASAA
S => ASA => aSA => aASAA => aaSAA
S => ASA => aSA => aASAA => aaSAA => aaASAAA
S => ASA => aSA => aASAA => aaSAA => aaASAAA => aaaSAAA
!! backtrack !!
S => ASA => aSA => aASAA => aaSAA => aaASAAA
!! backtrack a nouveau car pas d autre regle pour A !!
S => ASA => aSA => aASAA => aaSAA
S => ASA => aSA => aASAA => aaSAA => aaASBAA
S => ASA => aSA => aASAA => aaSAA => aaASBAA => aaaSBAA
!! backtrack !!
S => ASA => aSA => aASAA => aaSAA => aaASBAA
!! backtrack a nouveau car pas d autre regle pour A !!
S => ASA => aSA => aASAA => aaSAA
S => ASA => aSA => aASAA => aaSAA => aaAAA
S => ASA => aSA => aASAA => aaSAA => aaAAA => aaaAA
!! backtrack !!
S => ASA => aSA => aASAA => aaSAA => aaAAA
!! backtrack a nouveau car pas d autre regle pour A !!
S => ASA => aSA => aASAA => aaSAA
!! backtrack a nouveau car pas d autre regle pour S !!
S => ASA => aSA => aASAA
!! backtrack a nouveau car pas d autre regle pour A !!
S => ASA => aSA
S => ASA => aSA => aASBA
S => ASA => aSA => aASBA => aaSBA
... (on a compris l idee)
```

3.7 Packrat parsing (not in the book)

Pour éviter le coût du backtrack parsing (qui peut être exponentiel si à chaque fois on se retrouve à devoir prendre la dernière dérivation), on utilise de la mémorization. On obtient donc un algorithme en O(input*#production).

On va pour chaque index dans l'input essayer de parser via toutes les règles possibles et stocker le résultat ainsi que l'index de fin de l'expression. On peut ainsi aller de l'index le plus grand au plus petit en utilisant les résultats précedemment calculés.

4 Chapitre 4 : Type checking

4.1 Introduction

Du *type checking*, c'est faire de l'analyse sémantique. On y applique les opérations suivantes :

— Détermine le type de tous les noms et expressions.

— Type checking : on vérifie les types des opérations

— Analyse du stockage pour savoir la place reprise par les variables locales

— Réécriture de l'arbre AST pour rendre explicite certaines constructions.

4.2 j-- types

4.2.1 Introduction

On a soit des **types primitifs** (`int, boolean, char`), soit des **types référentiels** (`array`, type définit par une déclaration de classe, `java.lang.Object`, etc.).

4.2.2 Type representative and class objects

On défini deux objets *placeholder* pour prendre la place des types non résolu encore :

— Type Name : est résolu en faisant un lookup dans le contexte (représentation dy symbol table)

— Array Type Name : on résoud le type des éléments de l'array et on l'encapsule dans une *Class Object* utilisé ensuite dans l'array.

On utilise cette technique car le type défini ce qu'il nous faut et on peut utiliser des types temporaire avant la résolution.

4.3 j-- Symbol Table

4.3.1 Contexts And Idefns : Declaring and Looking Up Types and Local Variables

La table des symboles est représentée par un arbre d'objets contexts, chacun correspondant à une région de scope du programme et contenant un mapping des noms vers leur définition (type, parameter, local variables).

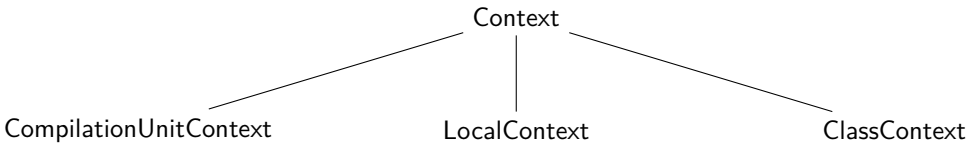


FIGURE 13 – Context tree

Dans chacun des contextes associés aux méthodes de classes, on associe un champ *surrounding context* qui pointe vers le contexte englobant le bloc correspondant.

Contexts A chaque point d'execution, la table des symbols ressemble à une **stack de context** :

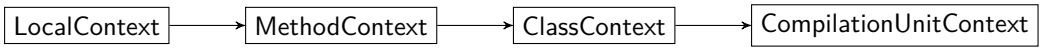


FIGURE 14 – Stack context at time t

Durant l'analyse, quand le compilateur rencontre une **variable** il regarde à cette variable dans la table des symbols en commencant par le `localContext`.

— `CompilationUnitContext` : scope du programme entier et contient un mapping des noms vers les **types** :

1. Les type implicits (java.lang.Object, java.lang.String,. . .)
2. Les types importés
3. Les types introduit dans la déclaration de classe

— `ClassContext` : scope d'une classe et ne contient pas de mapping. (Constructor, methodes et champs sont enregistré dans l'objet Class)

— `MethodContext` : scope d'une méthode dont les paramètres sont déclarés dedans

— `LocalContext` : scope d'un block ({ }) contenant les variables locales

Idefns est un type d'interface pour la définition de la table des symboles.

Deux types de Idefns dans les entrées de la table :

— `TypeNameDefn` qui encapsule juste un type

— `LocalVariableDefs` qui défini une variable locale en encapsulant son nom, type et un offset dans la frame stack.

4.3.2 Finding Method and Field Names in Type Objects

LN

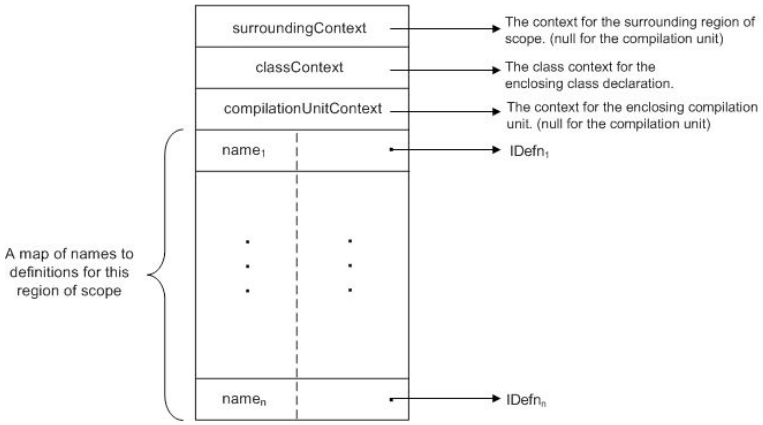


FIGURE 15 – Structure d'un context

4.4 Two phase analysis

L'analyse sémantique besoin de deux traversés de l'AST car des classes et méthodes peuvent être déclarées après une référence.

Globalement, **preAnalyze()** fait ComputationUnitcontext et ClassContext ainsi que les informations à propose des types importés/déclaré. **analyze()** fait MethodContext et LocalContext, continue la création des tables de symbol, réécrit l'arbre et fait plusieurs checking.

4.5 Pre-Analysis of j-- program

`PreAnalyze()` va :

— déclarer les types importés (`JCompilationUnit` level)

— déclarer les class définies par l'user (`JClassDeclaration` level)

— déclarer des champs (`JFieldDeclaration` level)

— déclarer les méthodes (`JMethodDeclaration` et `JConstructorDeclaration` level)

4.5.1 `JCompilationUnit.preAnalyse(Context c)`

— Créer le `CompilationUnitContext`

— Déclarer les types implicites `java.lang.String` et `java.lang.Object`

— Déclarer les types importés

— Déclarer les types définis par les déclarations de classe

— Invoquer `preAnalyze()` pour chacun des déclarations de type dans la Compilation Unit

4.5.2 `JClassDeclaration.preAnalyse(Context c)`

— Créer le *Class context* avec `surroundingContext` qui pointe vers `CompilationUnitContext`

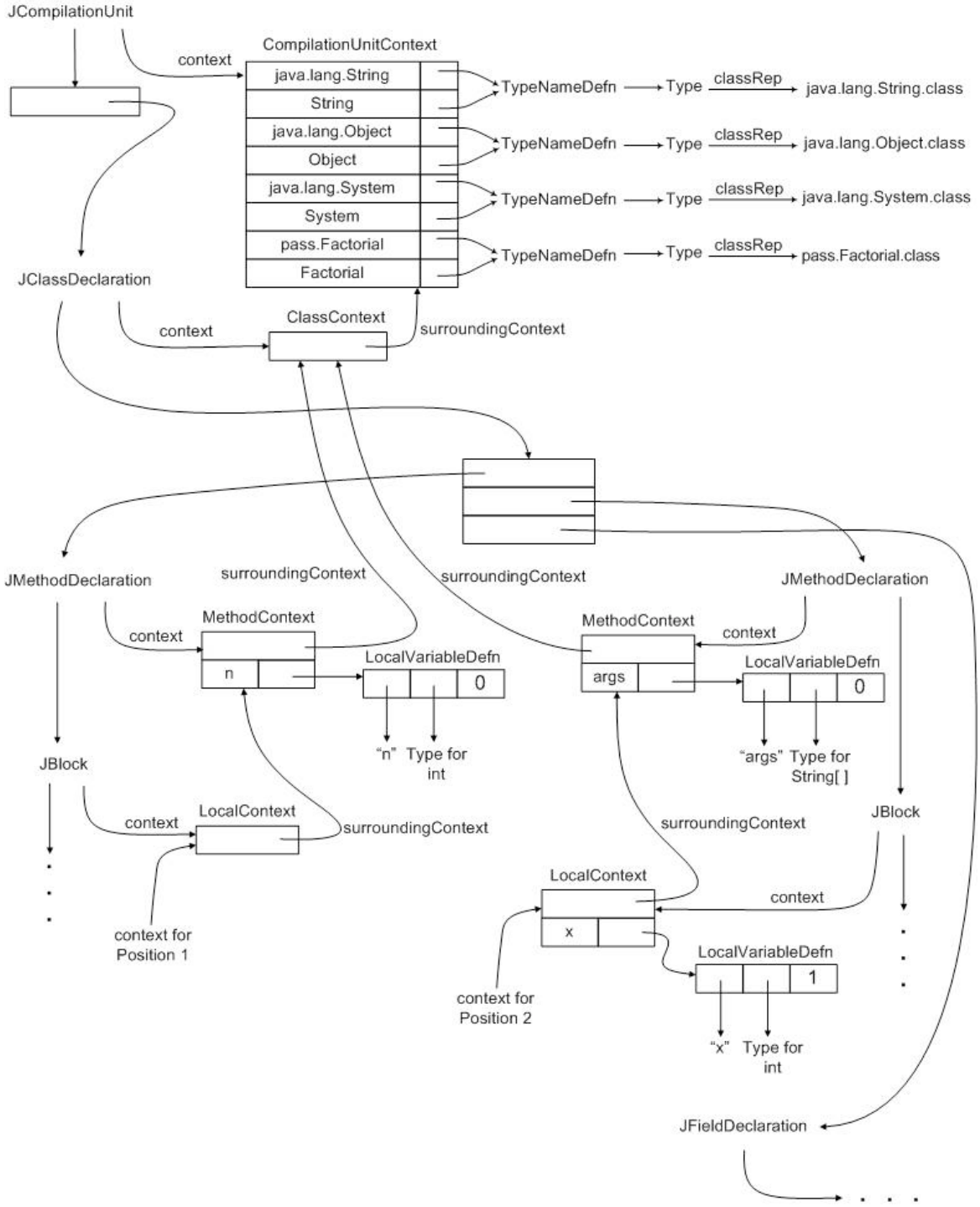
— Résoudre le type `super` de la classe (check si on n'étend pas un class *final*)

— Créer le `CLEmitter` associé et y ajoute le class header, le nom et tout les modifications

— Invoquer `preAnalyze` sur les membres de la classe

— Rajouter un constructeur explicite s'il n'y a pas

— Produire le *Class object*.




```
package pass;
import java.lang.System;

public class Factorial {
    public static int factorial(int n) {
        if (n <= 0) return 1;
        else return n * factorial(n - 1);
    }
    public static void main(String[] args) {
        int x = n;
        System.out.println(x + "! = " + factorial(x));
    }
    static int n = 5;
}
```

FIGURE 16 – Example AST with context

4.5.3 `JMethodDeclaration.preAnalyse(Context c, CLEmitter cle)`

- Résoud les types des paramètres et du `return`
- Vérifie si l'*abstract modifier* est propre
- Calcule du descripteur de méthode (codification de la signature en string)
  (`(I)I` : consomme un `int`, renvoie un `int` ; ([`Ljava.lang.String;`])V : prend une liste ([]) de strings (`Ljava.lang.String;`), revoie `void`)
- Appelle `partialCodeGen()` pour générer le code de la méthode sans le corp pour que l'objet class ait au-moins les informations de l'interface des méthodes (paramètre et le type du `return`)

4.5.4 `JFieldDeclaration.preAnalyse(Context c, CLEmitter cle)`

- Vérifie qu'un champs n'est pas déclaré `abstract`
- Résoud le type du champ
- Génère le code bytestream pour la JVM pour la déclaration des champs

4.5.5 Symbol Table built by `preAnalyse()`

L'étape de pré-analyse ne déclare aucune variable locale. On a donc un arbre partiel qui est construit.

4.6 Analayse of j-- programs

Lors de l'analyse, on va effectuer une discrete analyze recursive :

- Réécriture de champs et d'initialisation de variables locales comme des assignements de variables normales
- Déclaration des paramètre formet et des variables locales
- Allocation des ressources sur la stack frame
- Calcul des types des expressions
- Reclassification des noms ambigus
- Modification de l'arbre

4.6.1 `JCompilationUnit.analyse(Context c)`

Appelle analyze() pour descendre dans l'arbre pour chaque type de déclaration.

4.6.2 `JClassDeclaration.analyse(Context c)`

Après avoir fait l'analyze() pour chaque field, il copie l'initialisation des fields dans son node.
Il va aussi les séparer en deux : `staticFieldInitializations` et `instanceFieldInitializations`.

4.6.3 `JFieldDeclaration.analyse(Context c)`

Récrit le field initialisé comme un assignement explicit stocké dans la liste d'initialisation.

4.6.4 `JMethodDeclaration.analyse()`

1. Crée un **MethodContext** dont `surroundingContext` pointe vers le `ClassContext` précédant
2. Le premier stack frame offset = 0.
   (0 is allocate for `this` for a instance method)
3. Les paramètres sont déclaré comme des variables locales et alloué sur des offsets consécutif
4. Il analyse le corps de la méthode comme un context différent.

Stack-frame method Nous avons besoin de calculer les offsets de tout les paramètres ainsi que des variables local pour savoir l'espace nécessaire à la méthode.

4.6.5 `JBlock.analyse()`

1. Crée un nouveau `LocalContext` dont `surroundingContext` pointe vers le `MethodContext` précédent et le **newtOffset** est copié du précédent context.
2. Analyze son body et chaque `JVariableDeclarations` declare sa variable dans le `LocalContext` du step 1

4.6.6 Rewriting a field

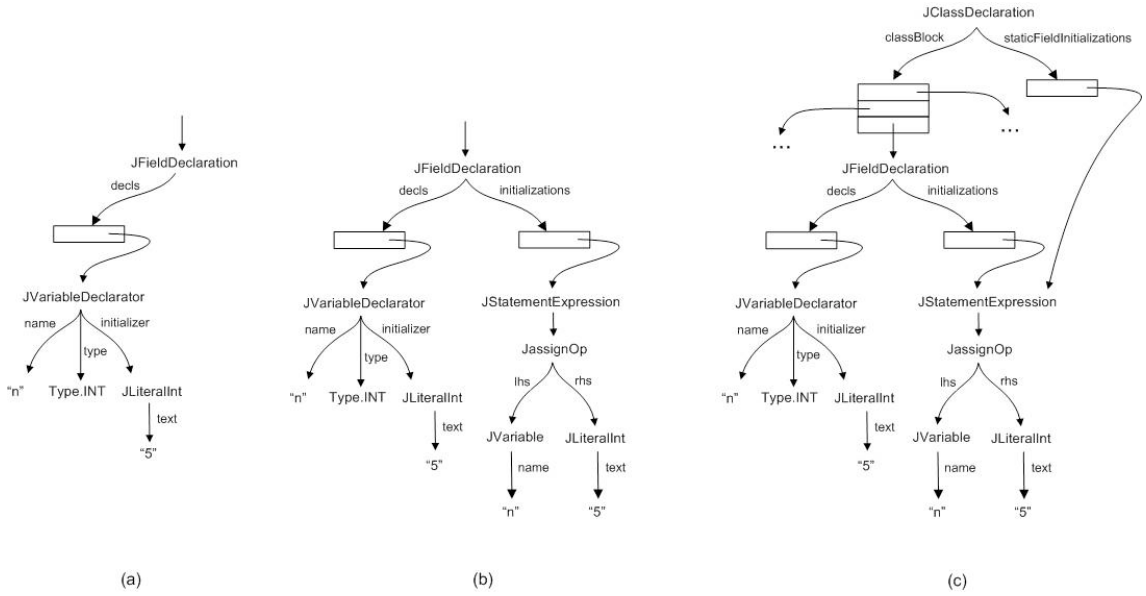


FIGURE 17 – Rewriting a field Initialization. JClassDeclaration (c) et JFieldDeclaration (b) analyse

4.6.7 Method context and local context

```
public class Locals {
    public int foo(int t, String u) {
        int v = u.length();
        {
            int w = v + 5, x = w + 7;
            v = w + x;
        }
        {
            int y = 3;
            int z = v + y;
            t = t + y + z;
        }
        return t + v;
    }
}
```

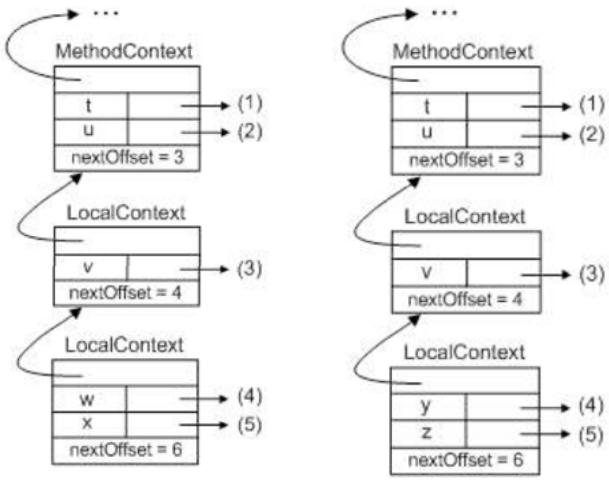


FIGURE 18 – Example method context and local context

4.6.8 Analysing local variable declaration and their initialization

Avant d'effectuer la méthode analyze(), l'AST contient uniquement les `JVariableDeclarator`. L'analyse va :

— Créer une `JVariableDefns` avec l'offset de la stack-frame correspondant

— Vérifié que la variable déclaré n'a pas un nom déja existant

— Déclaré la variable dans le `LocalContext`

— Réécrire tout les initialization comme des assignements explicits

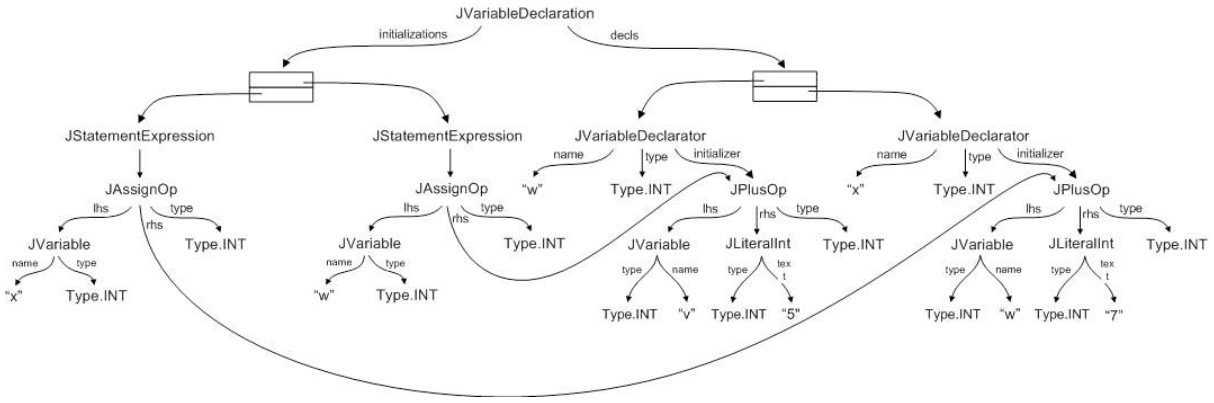


FIGURE 19 – Analysis variable initialization

4.6.9 Simple variable

— Recherche dans la table des symbols la variable et récupère son `LocalVariableDefn`

— Enregistre le `LocalVariableDefn` pour son utilisation dans le code generation et copie le Type

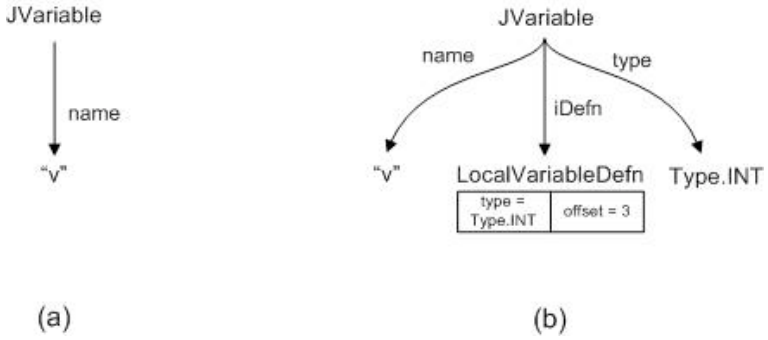


FIGURE 20 – Analysis simple variable

Note : Une variable peut représenter un field d'une classe, dans ce cas la variable est analyser différement.

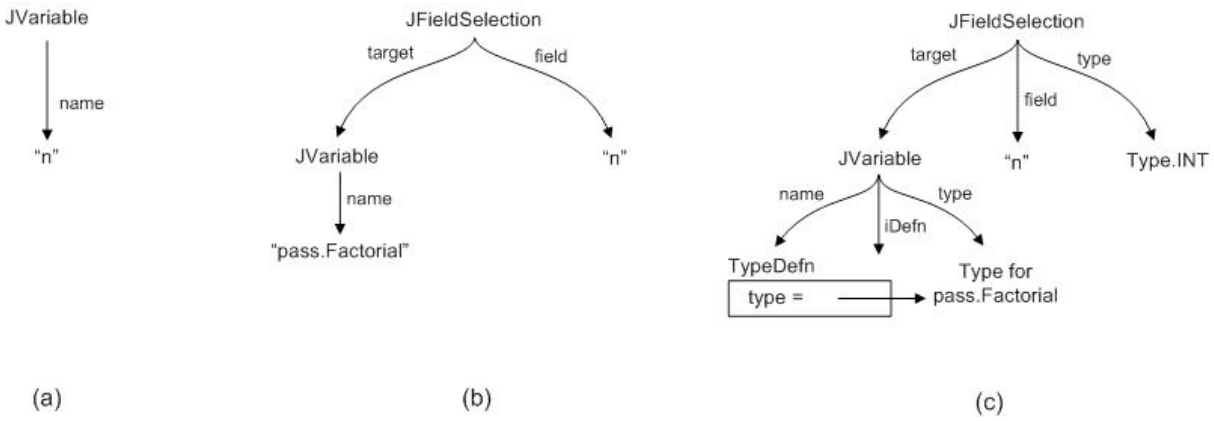


FIGURE 21 – Analysis d'un field

4.6.10 Field selection and message expression

4.6.11 Typing Expressions and Enforcing the Type Rules

4.6.12 Cast operation

4.7 The Visitor Pattern and the AST Traversal Mechanism

4.8 Visitor Pattern and the AST traversal Mechanism

5 Chapitre 5 : JVM and Bytecode

5.1 Introduction

Une machine virtuelle est un programme implémentant une machine qui exécute un programe comme une machine physique.

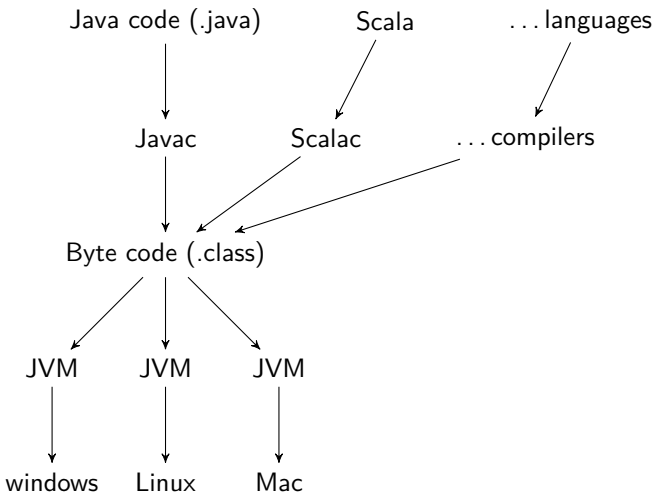


FIGURE 22 – JVM

D'autres language choisissent de rejoindre la JVM car c'est facile de générer le code et le byte code est compacte, le code est sur une architecture indépendante et donc compatible et finalement JVM est efficace et très testé !

Une instruction **bytecode** =
- Un byte **opcode**
- Un nombre variable d'argument
- Offset ou pointeur vers la Constant pool

Une instuction byte code consumme et produit certain élément de la stack. Constantes, locals et les éléments de la stack sont typé (adresse ou type primitifs).

Un fichier **.class** =
- Magic number (0xCAFEBABE)
- Version
- Constant pool
- Access flag
- This class
- Super class
- Interfaces
- Fields
- Methods
- Atrributes

5.2 Components

- Stack par thread : stocke les stack frames
- Heap : mémoire alloué dynamiquement
- Constant pool : stocke les constantes et les références vers les champs/méthodes
- Code segment : stocke code du programme (JVM instruction)
- Counter du programme par thread : contient l'addresse de la prochaine instruction

5.3 Stack

JVM a décidé d'utiliser une stack machine car cela réduit la taille du byte-code et surtout toutes les opérations travailles sur peu d'élément au dessus de la stack !

(*Pour une operation de division d'entier, la machine RISC à 32 registres on a besoin* $32 * 31$ *différents opcode (pour chaque paire possible de registre), alors que la stacke machine n'a besoin que de 1 opcode puisqu'il travaille toujours sur la deux éléments en haut de la stack*)

5.3.1 Stack loading process

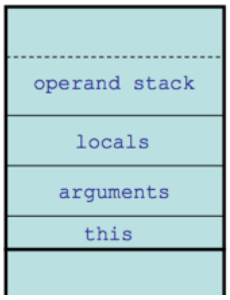
Les classes (dont le nom doit correspondre avec celui du fichier) sont chargées de manière *lazy* lors du premier accès, juste après les super-classes dont elles dépendent. Ensuite le byte-code est vérifié, les fiels static sont alloué et recoivent leur valeur par défault. Enfin, les initialiseur static sont exécuté.

5.3.2 Frame

Une nouvelle frame est créé et push sur la stack pour chaque appel de méhtode. Les stacks sont ensuite pop lorsque la méthode se termine normalement ou si une exception est *throw*.

Une frame =

— Un tableau de variable locale
— Une valeur de retour
— Operand stack
— Référence vers le `runtime constant pool` pour la classe de la méthode courante.



5.3.3 Operand stack

Appliquer des opérations se fait sur des éléments en haut de la stack. Dans cet example deux éléments sont consommé et un produit.

```
public int add(int v1, int v2) {
    int c = 8;
    return v1+v2+c;
}
```

→

```
0 bipush 8
2 istore_3 [c]
3 iload_1 [v1]
4 iload_2 [v2]
5 iadd
6 iload_3 [c]
7 iadd
8 ireturn
```

FIGURE 23 – Exemple d'utilisation des opérations de stack

5.4 JVM instruction

Arithmetic `ineg, i2c, iadd, isub, imul, idiv, irem, iinc k i`

Branch `goto L, ifeq L, ifne L, ifnull L, ifnonnull L` : compare l'élément au top de la stack par rapport à 0.
`if_icmpeq L, if_icmpne L, if_icmplt L, if_icmple L, if_acmpeq L, if_acpmne L, if_icmgt, if_icmpge` : compare les 2 éléments au top de la stack

Constant loading `iconst_0, ..., iconst_5, aconst_null, ldc i, lds string`.

Field `getfield f sig, putfield f sig, getstatic f sig, putstatic f sig`

Stack `dup, pop, swap, nop, dup_x1, dup_x2`

Class `new C, instanceof C, checkcast C`

Méthode `invokestatic name sig, invokevirtual name sig, ireturn, areturn, return`

5.5 Verification

Un bytecode nécessite d'être vérifié avant son exécution. Cette vérification est effectué : au chargement de la class et à l'exécution (runtime).

Note : Un compilateur java doit générer du code vérifiable

5.5.1 File

Grace au code magique (0xCAFEBABE)

5.5.2 Constant and header

Les classes final ne sont pas hérité, final méthode non réécrite, chaque classe à une superclass (sauf Object), field et méthode référence a signature valide.

5.5.3 Instruction

Seulement les offset légaux sont référencé, les constantes ont des types appropriés, toutes les instructions sont complète.

5.5.4 Dataflow and type checking

5.6 CLEmitter

6 Garbage Collection

— Explicit memory management
  1. Peut amener à beaucoup d'erreur (double free/malloc, seg fault,...)
  2. Programmation modulaire plus complexe (accord sur qui free/malloc,...)
  3. trouver de corriger les bugs est très compliqué car parfois un bug apparait longtemps après le lancement du programme.

— Automatic memory management
  1. Illusion d'une infinité de mémoire

Allocation

— On maintient plusieurs **freelist** tel que `freelist[i]` contient les records de taille i

— Lorsqu'un objet demande un chunk, on peut donner n'importe quel block de taille $>=$ celle demandée.

6.1 Reference Counting

— Chaque objet retient le nombre de référence vers lui-même.
  → `cnt++`/`cnt-` lors decréation/destruction de référence

— Suppression lorsque le `cnt == 0`

\+ Méthode rapide, sûre et marche dans la plupart des cas.

\- Problème si il y a des références circulaire (A ref B et B ref A)

6.2 Mark & Sweep

1. **Mark** : On démare de root, on parcour la mémoire comme un graph (object graph) en marquant tout les objets atteignables à partir des roots.
   — Complexité temporel : proportionnel au number de node mark.
   — Complexité spatial : Puisque DFS est recursive, la complexité pour un heap de size H peut être de $\mathcal{O}(H)$. → Pas acceptable

Nom	Avantage/inconvénient
Reference counting	+ Rapide, sûre - Ne fonctionne pas avec des ref circulaires
Mark & sweep classique	- Complexité spatiale $\mathcal{O}(H)$, fragmentation et coût d'allocation
Mark & sweep reversibel	+ La stack est "contenue" dans le graph pointer - Fragmentation et coût d'allocation
Copying collection	+ allocation facile et peu couteuse, pas de fragmentation (compactage automatique), le temps est proportionnel au nombre d'objets vivants → efficient - Il faut le double d'espace mémoire.
Generation collection	Peut être utilisé pour — Mark & sweep — Copying

TABLE 3 – Garbage solution

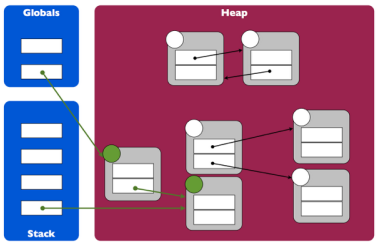
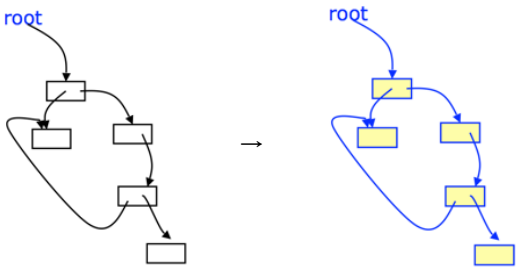


FIGURE 24 – Mark & sweet principe

2. **Mark avec Pointer reversal** :

l'idée est de faire un DFS et d'encoder la stack dans le graph pointer lui-même en inversant les pointeurs.




```
function DFS(x)
    if x == pointer AND record x not marked
        t = nil
        mark x
        done[x] = 0
```

```
function DFS(x)
    if x == pointer into the heap AND
         record x is not marked
        mark x

        foreach field f_i of record x
            DFS(x.f_i)
```

```
function DFS(x)
    if x == pointer into the heap AND
         record x is not marked
        mark x

        t=1
        stack[t] = x

        while t > 0
            x = stack[t--]

            foreach field f_i of record x
                if x.f_i == pointer AND record
                      x.f_i not marked
                    mark x.f_i
                    stack[++t] = x.f_i
```

FIGURE 25 – Implementation mark phase sans re

```
        while true
            i = done[x]

            if i < # of fields in record x
                y = x.f_i

                if y == pointer AND record y not marked
                    x.f_i = t
                    t = x
                    x = y

                    mark x
                    done[x] = 0
                else
                    done[x] = ++i
            else
                y = x
                x = t

                if x = nil then return
                i = done[x]
                t = x.f_i
                x.f_i = y
                done[x] = ++i
```

3. **Sweep** : on itère sur tout les objets du heap, on supprime les objets non marqué et on clear les marks.

```
function DFS(x)
    p = first address in heap
    while p < last adress in heap
        if record p marked
            unmark p
        else %% (let f_1 the first field in p
            f_1 = freelist
            freelist = p

        p = p + sizeof(p)
```

FIGURE 26 – Implementation mark phase

Le mark and Sweep permet de gérer les cycles et ne déplace pas les objets dans la mémoire ce qui lui permet d'être applicable à des langages comme C/C++. Par contre, il peut y avoir des problèmes de fragmentation de la mémoire et un cout d'allocation élevé.

6.2.1 Conclusion

On peut le faire en DFS mais contraignant pour la complexité spatiale donc une meilleur solution est d'utiliser l'idée du pointer reverse.

Le pointer reverse est utilisé pour ne pas utiliser de la mémoire lors du garbage collection, en effet on lance en général un garbage collector quand on est a court de mémoire alors si il doit lui-même utiliser de la mémoire pour s'exécuter il va avoir un problème.

De plus l'espace mémoire requis pour le garbadge collector avec Mark & Sweep n'est pas borné, il dépend du nombre d'éléments a concerner.

Garbage Le cout peut être très élevé si il y a peu à supprimer.

Une meilleur solution est de mesuré H (heap size) and R (reachable data in heap) tel que si $\frac{R}{H} > 0.5$ on augmente la taille du heap.

$$\text{Cost} : \frac{c_1 R + c_2 H}{H - R}, \quad \text{avec c constant number of instructions}$$

→ Le mark and Sweep permet de gerer les cycles. Par contre, il peut y avoir des problème de fragmentation et un cout d'allocation grand.

6.3 Copying collection : Cheney algorithm

On splitte la heap en 2 : from-space (old space) et to-space (new space)

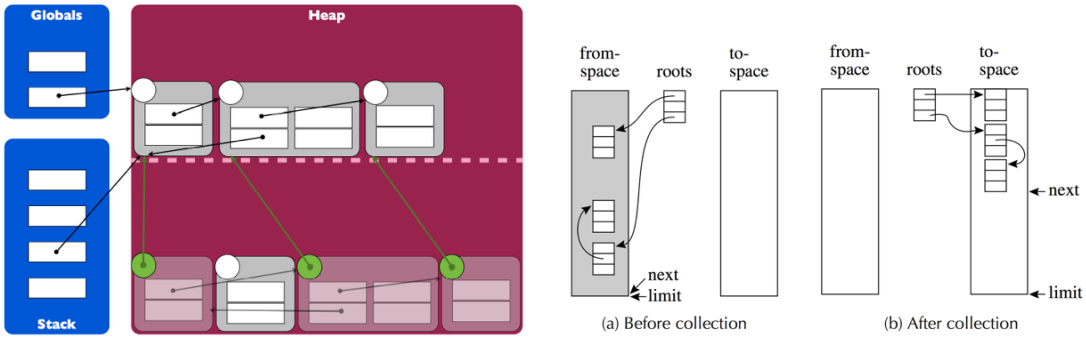


FIGURE 27 – Cheney principle

— Pour les references venant de la stack :

1. Si l'objet référencé est dans l'espace from, il faut l'amener dans l'espace to et laisser un pointer qui pointer vers le nouvel emplacement (forwarder pointer).
2. Si l'objet référencé est déjà dans l'espace to, il faut mettre à jour la référence à partir du forwarder pointer.

— Pour les références venant de l'espace to, on effectue les même opération que au dessus.

Etapes

1. Initialement tout les chunck utilisé sont dans le from-space.
2. Au lieu de marquer les chuncks utilisés, on les copie dans le to-space tout en maintenant à l'ancien emplacement une référence vers le nouvel emplacement.

→ On déplace ainsi petit-à-petit en suivant les liens.

3. Lorsque l'algo est fini, on inverse les utilisations des deux espaces (from devient to et to deviens from).

```
function Forward(p)
    if p points to from-space
        if p.f1 points to to-space
            return p.f1
        else
            foreach field fi of p
                next.fi = p.f i
            p.fi = next
            next = next + size of recod p
            return p.fi

    else
        return p
```

```
scan = next = beginning of to-space

foreach root r
    r = Forward(r)

while scan < next
    foreach field fi of record at scan
        scan.fi = Forward(scan.fi)

    scan = scan + size of record at scan
```

FIGURE 28 – Cheney algorithm

6.3.1 Avantages/inconvenient

— Simple : pas besoin de stack

— Allocation facile et peu couteuse : `malloc(n)` implémenté avec `next = next + n`

— Pas de fragmentation (compactage automatique)

— Le temps est proportionnel au nombre d'objets vivants.

— le garbage collection des objets est également peu couteux spécialement si beaucoup d'objet sont à jeter

— Mais vu que ça déplace les objets dans la mémoire certain langages ne le suportte pas C/C++

Par contre, il faut le **double** d'espace mémoire.

6.3.2 Remarque

Par contre le même problème qu'avec mark and sweep sans la technique des pointeur inversé se pose. Il faut traverser tous les elements sans utiliséde l'espace mémoire pour le faire.

Pour cela le to-space (new space) va être divisé en 3 parties :

— copier et scanner : les pointeurs contenu dans l'objet ont été scanné et "suivit"

— copier non scanner : les pointeurs contenu dans l'objet n'ont pas été traité

— empty

Etapes :

1. Copier dans le to-space les objets pointés par les roots et de créer les "forwarding pointers"
2. Scanner les objets non scanner et copier tous les objets pointer par ceux ci dans les to-space

→ Si on rencontre un objet déjà copié il suffit de suivre son forwarding pointer pour savoir ou il se trouve.

6.4 Generational collection

On divise la heap en deux ou plusieur parties, les jeunes et les vieux objets. Quand on alloue de la place pour un objet il est mis dans la partie jeune.

— On ne fait du garbage collection que dans les objets concidéré comme "jeune" a un rythme **soutenu**.

— Les vieux objets sont ceux qui ont survécu à plusieurs passage du garbage collector. Le rythme est plus lent.

— En effet, la majorité des objects meurent "**jeune** et ceux qui reste atteignable pour longtemps sont souvent : global, referencé depuis la main

⇒ On a alors une siscion dans la mémoire entre jeune et vieux. Lorsqu'un objet devient vieux, il est déplacé dans la partie vieux.

Un problème survient quand on a des objets dans la partie vieux et que ils référence des objets dans la partie jeune. Quand on va garbage collection les jeunes, on ne va pas voir que un objet est référencé par un "vieux". Pour cela on crée un table (remember set) qui contient tous les pointeurs des vieux vers les jeunes.

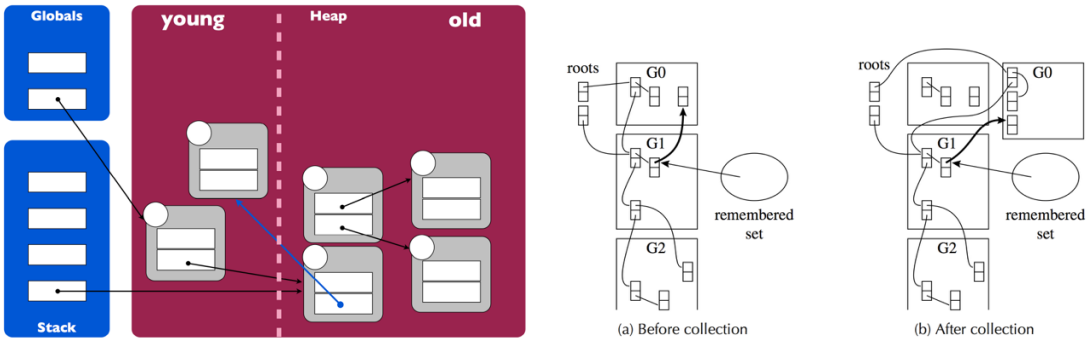


FIGURE 29 – Generation principle

7 DSL & Scala

7.1 Basic

7.1.1 Variables

Mot-clef

- `val`
- `var`
- `lazy val` : evalué uniquement au premier accès

```
var jizz = "jizz"
jizz = "baufays"        // OK

val poulpe = "poulpy"
poulpe = "manteau louche" // NOP
```

Type

```
val sum = 1 + 2 + 3
val nums = List(1, 2, 3)
```

Type inference

```
val sum: Int = 1 + 2 + 3
val nums: List[Int] = List(1, 2, 3)
```

Explicit types

7.1.2 Mutability Vs non-mutability

On encourage val plutôt que var, ainsi que des collections immutable.

7.1.3 Pattern matching

```
val times = 1
times match {
    // On value
    case 1 => ...

    // With guards
    case i if i == 2 => ...

    // On types
    case j: Int => ...
    case d: Double if d > 0.0 => ...

    // Default
    case _ => ...
}
```

```
object Test extends App {
    def bookType(b: Book) = b mathc {
        case Book(5) => ''Revue''
        case Book(3) => ''Journal''
        case Book(i) => ''default: %s''.format(i)
    }
}
```

7.1.4 Imports

Les imports peuvent apparaitre n'importe où dans le code.

7.1.5 Exception

`Nothing` est un subtype of tout les autres types.

```
def error(message: String): Nothing =
    throw new RuntimeException(message)

def divide(x: Int, y: Int): Int =
    if (y != 0) x / y
    else error(``cannot divide by zero'')

// Return Int est respecte comme Nothing est un subtype de Int
```

7.2 Orienté object

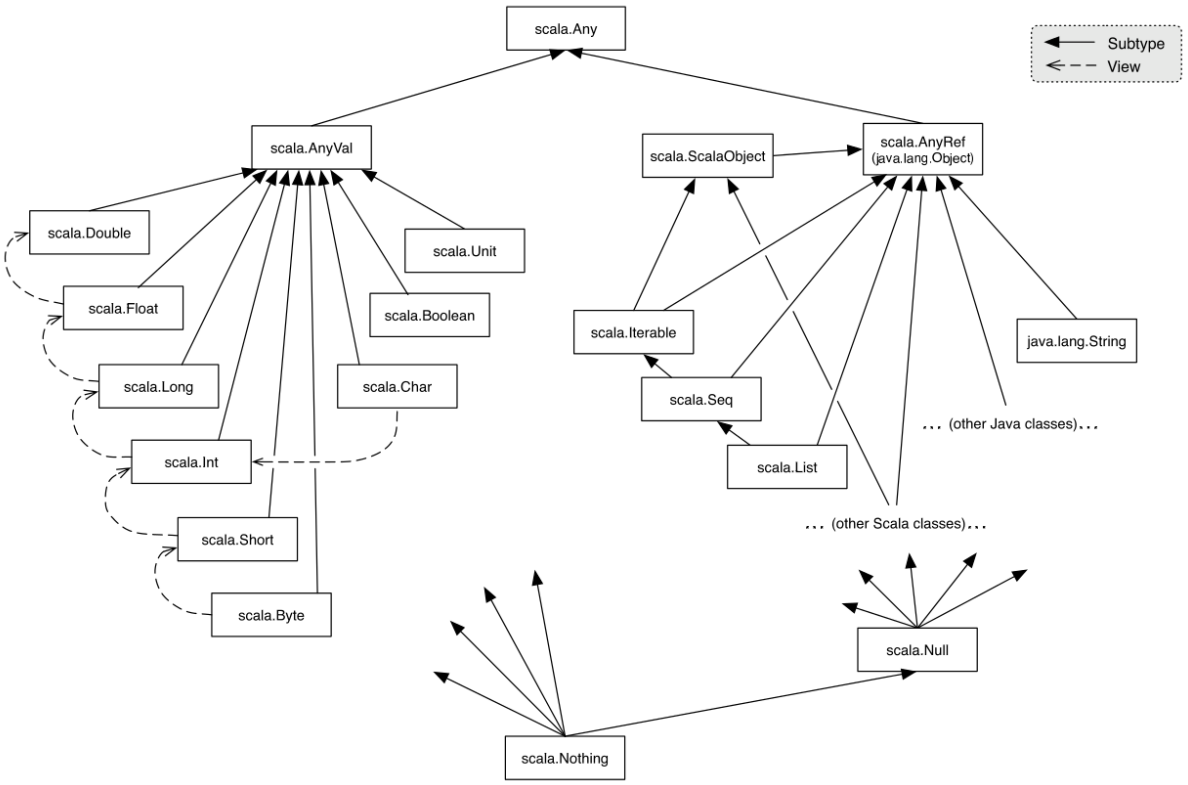


FIGURE 30 – Hierarchie

7.2.1 Pure O.O.

- Tout est un object : `1.toString`
- Toute opération est un appel de méthode : `1 + 2 + 3` → `(1).+(2).+(3)`
- Can omit . and ()

7.2.2 classes

App est utilisé comme la classe main en Java.

Les constructeurs

```
class vehicle(name: String, price:Int) {

}

val car = new vehicle("Ferrari", 100000);
```

Constructeur simple par défaut

```
class truck(name: String, price:Int) {

    def this(name: String) = {
        this(name, 10)
    }
}
```

Constructeur custom

object Les `object` sont comme des classes statiques. Ils représentent la déclaration d'une classe anonyme qui étend App ainsi que la création d'une unique instance "Test" de cette classe anonyme.

```
object Test extends App
```

→ Les méthodes définies à l'intérieur d'un object sont des méthodes statiques, au sens #Java8 du terme.

getters and setters Pas de getter/setter en scala, on y accède comme un champs : `nomDeLaClasse.Champ = 'Hélène roule en tracteur'`

Il y a aussi une autre technique si vos attributs sont privé vous pouvez les accéder comme ceci :

```
private var battery = 0
def power = battery // getter
def power_= (p: Int) { battery = p } // setter
```

la ligne power_= est un peu particulière, cela permet de définir un nom de fonction comprenant un espace. "power ="

Companion object and apply Un objet compagnon est un objet qui porte le même nom que la classe qu'il accompagne.

La spécificité étant que la classe peut accéder au membres privés de l'objet et vice versa. Le compagnion object peut être utilisé comme un genre de container.

`apply` est une méthode qui agit un peu comme un constructeur pour les objects et qui peut être utilisée avec les classes comme sucre syntaxique :

```
//Exemple 1

class Calculator(brand: String) {
       def add(m: Int, n: Int): Int = m + n
}

object Calculator {
       def apply(brand: String) = new Calculator(brand)
}

object Test extends App {

val c1 = new Calculator("HP") // Utilise le constructeur de Calculator
val c2 = Calculator("HP")   // Utilise apply() du compagnon
}
```

Un compagnion object permet de construire un object sans utiliser de new.

Operator overload Il suffit de réecrire une méthode avec le même nom.

7.3 Functional

7.3.1 Fonction anonymes

```
val plusOne = (x:Int) => x+1

plusOne(5)   // va retourner 6
```

7.3.2 Closure

Function anonyme qui peut référencé n'importe quelle valeur/variable dans le scope.

```
//     plusFoo can reference any values/variables in scope
var foo = 1
val plusFoo = (x:Int) => x + foo

plusFoo(5)   //return 6
foo = 5
plusFoo(5)   //now return 10
```

7.3.3 Higher order function

1. L'utilisation de _
2. Une fonction peut prendre une autre fonction en paramètre.

```
val nums = List(1,2,3)

nums.exists(_ == 2)   // true
nums.map(x => x+1)    // List(2,3,4)
```

7.3.4 Currying

On peut définir plus qu'une séquence d'arguments.

```
def add(a: Int)(b: Int) = a + b

// Partially applied function
val f = add(1)(_)
```

```
def add(a: Int) = (b: Int) => a + b
```

7.4 Null and options

La valeur `null` est difficile à gérer pour empecher le risque de `NullPointerException`.

`Option[T]`

— Some(x) if x existe

— None sinon

```
trait Option[T] {
    def isDefined: Boolean
    def get: T
    def getOrElse(t: T): T
}
```

7.5 Call by name

Il existe deux types de passage d'argument :

— **By value** : Les arguments sont toujours évalué AVANT l'exécution de la fonction de la même manière que les opérateurs.
L'idée est de réduire une expression à une valeur.

→ Si on utilise une fonction en argument elle ne sera évalué qu'une seule fois

— **By name** : Les arguments sont évalués lorsque l'on en a besoin. (≈ *lazy*)

→ Si on utilise une fonction en argument elle sera évalué à chaque appel (jamais évalué si on ne l'utilise pas).

```
def foo(y: => Int): Int = y
```

Compilé en →

```
def foo(y: () => Int): Int = y
```

By name est un sucre synthaxique pour la définition de closure

— Si CBV se termine ⇒ CBN se termine aussi.

Default En scala, le **call by value** est le comportement par défaut, et pour faire du **call by name** il faut ajouter ⇒ devant le type de l'argument.

7.5.1 Fonctionnement

Les deux méthodes sont équivalentes dans le cas de fonctions pures et si les exécutions terminent. Par exemple :

```
object Test {
   def main(args: Array[String]) {
        delayed(time());
   }

   def time() = {
      println("Getting time in nano seconds")
      System.nanoTime
   }
   def delayed( t: => Long ) = {
      println("In delayed method")
      println("Param: " + t) //1
      wait(5000)
      println("Param: " + t) //2
   }
}
```

Ici, si on fait du by value, les deux print imprimeront la même chose. Par contre, dans le cas du by name, la fonction sera recalculée et on aura deux valeur différente à imprimer. correct ?

7.5.2 Stream

Un stream est une liste dans laquelle le **tail** n'est executé que lorsque l'on en a besoin.

```
abstract class MyStream {
  def isEmpty: Boolean
  def head: Int
  def tail: MyStream

  def apply(i: Int): Int = {
    i match {
      case 0 => head
      case i => tail(i-1)
    }
  }

  def filter(p: Int => Boolean):
       MyStream = {
    if( p(head) )
      MyStream.cons(head, tail.filter(p)
          )
    else
      tail filter(p)
    }
  }
}
```

```
object MyStream {
  def cons(hd: Int, tl: => MyStream) : MyStream = new
       MyStream {
    def isEmpty = false
    def head: Int = hd
    def tail: MyStream = tl
  }

  val empty = new MyStream {
    def isEmpty = true
    def head = throw new NoSuchElementException("empty.head"
         )
    def tail = throw new NoSuchElementException("empty.tail"
         )
  }
}
```

Utilisation :

```
object TestStream extends App{
  def streamFrom(from: Int): MyStream =
    MyStream.cons(from,streamFrom(from+1))

  def streamFilter(from: Int, f: (Int) => Boolean): MyStream =
    streamFrom(from).filter(x => f(x))
```

```
  def nextPrime(from: Int, i: Int): Int =
    streamFilter(from, (x) => isPrime(x)) (i)

  def isPrime(i: Int): Boolean = {
    if (i <= 1) false
    else if (i == 2) true
    else !(2 to (i - 1)).exists(x => i \% x == 0)
  }
}
```

7.5.3 Flow-control

```
def myLoop(n: Int)(body: Unit): Unit = {
def myLoop(n: Int)(body: () => Unit): Unit = {
def myLoop(n: Int)(body: => Unit): Unit = {

    if (n > 0) {
        println(n)

        body
        body()
        body

        myLoop(n-1) (body)
    }
}

myLoop(5) {
myLoop(5) { () =>
myLoop(5) {

    println("hello")
}
```

Résultat

```
hello
5
4
3
2
1
```

```
5
hello
4
hello
3
hello
2
hello
1
hello
```

```
5
hello
4
hello
3
hello
2
hello
1
hello
```

7.5.4 Storing by-name parameters

```
class Observable {
    private var listeners: List[() => Unit] = List()
    def changed(): Unit = listeners.foreach(l =>l())
    def onChange(b: => Unit): Unit = {
        listeners = (() => b) :: listeners
    }
}
```

Example

```
class Counter() extends Observable {
    private var c: Int = 0
    def value = c
    def incr(): Unit = {
        c += 1
        changed()
    }
}
```

```
val c = new Counter()
c.onChange {
    println(c.value)
}
c.incr()
c.incr()
```

7.6 Implicit

7.6.1 implicit conversion

Permet de convertir automatiquement un type en un autre pour pouvoir appliquer certaines opérations.

```
import ComplexImplicits._

object ComplexImplicits {
       implicit def Double2Complex(value : Double) = new Complex(value,0.0)
}

object MyComplex extends App {
       val d = Complex(1)(2)
       val e = 1 + d //Ici, le Int 1 va automatiquement etre converti en Complex
                  //via la methode Double2Complex
}
```

7.6.2 implicit classes

Une classe implicite permet d'enrichir une classe existante en y ajoutant des méthodes comme par exemple :

```
object TestImplicits extends App {
implicit class PrimeTester(i: Int) {
    def isZero: Boolean = i == 0
}

3.isPrime
```

Le type string a maintenant une nouvelle méthode, method1 qui ne fait ... rien ! mais c'est quand même la classe.

7.6.3 implicit parameters

```
def speakImplicitly (implicit greeting : String) = println(greeting)
speakImplicitly("Goodbye world")
speakImplicitly // error: no implicit argument matching parameter type String was found.
implicit val hello = "Hello world"
speakImplicitly
```

Le keyword implicit va dire au compilateur de lui-même procurer un paramètre à la méthode. Pour cela il faut qu'il y ai un paramètre implicit déclaré et visible par la fonction et qu'il ai le type requis.

7.7 Traits

Les traits sont à la frontière entre les interfaces et les classes abstraites.

\+ interface

— Contient des implementation ⇒ plus riche que les interfaces

\- abstract

— Pas de paramètre dans le constructeur

— Permet un héritage multiple car il est lié dynamiquement

⇒ Ce n'est pas un object et on ne peut donc pas l'instancier

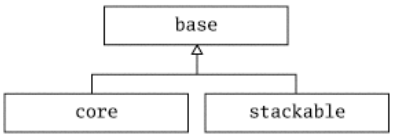
```
trait Comp {

def < (that : A): Boolean
def > (that : A): Boolean = {...}
def >= (that : A): Boolean = {...}
...
}
```

7.7.1 Stackable trait

— Le trait (ou classe abstraite) de **base** défini une interface abstraite que tout les cores ou stackables extend

— Le trait (ou classes) de **core** implémente les méthodes abstraite

— Chaque **stackable** override un ou plusieurs méthodes définie dans le **base** trait.




```
class Animal
trait Furry extends Animal
trait HasLegs extends Animal
trait FourLegged extends HasLegs
class Cat extends Animal with Furry with
      FourLegged
```

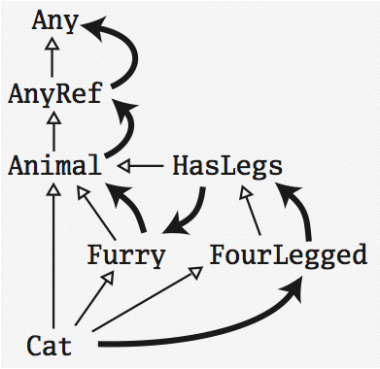


FIGURE 31 – Linearisation des héritages

7.8 Monads

Tous les types `M[T]` donc en gros des listes de `T` (structure de données génériques) et qui ont deux opérations spéciales : `flatMap` et `unit`

Il s'agit d'une structure de donnée M qui a un types paramétrique $M[T]$ qui défini :

— `flatMap` : idem que map mais avec un flatten en plus.

$$(\texttt{[1, [2]]} \Rightarrow \texttt{[1, 2]})$$

— `unit` : Prend une valeur en argument (du type générique de la structure) et va créer une structure avec cette élément dedans.

Ex : `List(x)` : on aura une liste avec `x` comme premier élément de la structure.

— `map` : prend un iterable et applique la fonction passée en argument

```
trait M[T] {
    def flatMap[U](f: T => M[U]): M[U]
    def map[U](f: T => U): M[U]

    def withFilter[U](p: T => Boolean): M[T]
    def foreach(f: T => Unit): Unit
}

def unit[T](x: T): M[T]
```

```
val l : List[Option[Int]] = List( Some(5), None, Some(7), Some(8))

// Link with for( ; ) yield
println( l map(_ map(x => x)) flatten)
println( l flatMap(_ map(x => x)))
println( l flatMap(_ flatMap(x => Some(x))))
println( for(i <- l; j <- i) yield j)
```

```
=> List(5, 7, 8)

// Link with for( ) yield
println( l map(x => x))
println( l flatMap(x => Some(x)))
println( for(i <- l) yield i)

=> List(Some(5), None, Some(7), Some(8))

// Link with for
println( l foreach(x => print(x)))
println( for(i <- l) print(i))

=> Some(5)NoneSome(7)Some(8)()
```

7.8.1 map - flatmap

```
m map f == m flatMap ( x => unit(f(x)) )

m flatMap f == flatten( m map f)
```

7.8.2 Monad law

Associativity

```
m flatMap f flatMap g == m flatMap (x => f(x) flatMap g)
```

Left unit

```
unit(x) flatMap f == f(x)
```

Right unit

```
 m flatMap unit == m
```

Example :

```
// Monads Laws
val m : List[Int] = List(1, 2, 3, 4, 5)
val x = 5
val f = (i : Int) => List(i * 2)
val g = (i: Int) => List(i + 10)

// Associativity
println ( m.flatMap(f(_)).flatMap(g(_)) == m.flatMap( f(_).flatMap(g(_))) )

// Left Unit
println ( List(x).flatMap(f(_)) == f(x) )

// Right Unit
println ( m.flatMap(List(_)) == m )
```

7.8.3 Exercice

```
  def sequenceOption[A](x: List[Option[A]]): Option[List[A]] = {
    x match {
      case Nil => Some(Nil)
      case h :: t => h flatMap (x => sequenceOption(t) map (x :: _))
    }
  }
```

7.9 Yield

Va faire une espère de link dans le type du premier du for :

```
val A = Array(1,2)
val B = List(1,2)

// Array((1,1), (1,2), (2,1), (2,2))
for (i <- A; j <- B) yield (i,j)

// List((1,1), (1,2), (2,1), (2,2))
for (i <- B; j <- A) yield (i,j)
```

7.9.1 Translation

```
for (x <- expr1) yield expr2

for (x <- expr1 if expr2) yield expr3

for (x <- expr1; y <- expr2; seq) yield expr3
```

→

```
expr1.map(x => expr2)

for (x <- expr1 withFilter (x => expr2)) yield expr3

expr1.flatMap(x => for (y <- expr2; seq) yield expr3)
```

7.10 Try

```
abstract class Try[+T]
case class Success[T](x: T) extends Try[T]
case class Failure(ex: Exception) extends Try[Nothing]

object Try {
    def apply[T](expr: => T): Try[T] =
    try Success(expr) catch {
        case NonFatal(ex) => Failure(ex)
    }
}

abstract class Try[T] {
    def flatMap[U](f: T => Try[U]): Try[U] = this match {
        case Success(x) => try f(x) catch { case NonFatal(ex) => Failure(ex) }
        case fail: Failure => fail
    }

    def map[U](f: T => U): Try[U] = this match {
        case Success(x) => Try(f(x))
        case fail: Failure => fail
    }
}
```

Example :

```
def divide: Try[Int] = {
    val dividend = Try(Console.readLine("Enter an Int to divide:\n").toInt)
    val divisor = Try(Console.readLine("Enter an Int to divide by:\n").toInt)
    val problem = dividend.flatMap(x => divisor.map(y => x / y))

    problem match {
        case Success(v) =>
            println("Result of " + dividend.get + "/" + divisor.get + " is: " + v)
            Success(v)
        case Failure(e) =>
            println("You must've divided by zero. Try again!")
            println("Info from the exception: " + e.getMessage)
            divide
    }
}
```