Développement d'un générateur d'interpréteur de bytecodes pour une JVM embarquée

Mustapha Tachouct

23 juin 2004

Remerciements

Tout d'abord, je voudrais remercier Marie-Claude Delecour pour l'immense travail qu'elle fournit chaque année dans le recherche de stage pour les étudiants.

Et surtout Alexandre Courbot, pour m'avoir proposé ce stage. Et aussi Moulay-Driss Benchiboun, mon responsable de stage de l'IUT. Ainsi que Gilles Grimaud pour ses précieux conseils. Je remerçie également l'équipe RD2P (muf, kartoche, goyan, david, pb, mickey, ...) pour sa sympathie.

// Abstract

I have been doing my trainee period in the LIFL research center for ten weeks. Within RD2P team, I have discovered the world of R & D about small devices.

This trainee about embedded systems included Java and C programming languages and a few assembly language. I worked on JITS project.

JITS is a Java-Based operating system designed for embedded-system, such as smart-cards. JITS aims to provide a full-featured JVM and a complete API for small devices without sacrificing functionnalities offered by Java. Indeed, this JVM-OS can be customized to fit the needs of the applications. In comparison to the other common implementations, JITS doesn't impose langage restrictions.

I first propose you to discover the context where this trainee took place, then the caracteristics of the JITS project. Finally, I will introduce my work : a interpeter generator.

Table des matières

1 Présentation du LIFL **6**

1.1 Introduction 6

1.2 Axes et équipes 7

1.3 L'équipe RD2P 7

2 Java In The Small **8**

2.1 Différentes versions et différentes VM 8

2.1.1 CDC 8

2.1.2 CLDC 8

2.1.3 JavaCard 9

2.2 JITS : Java In The Small 9

3 Générateur d'interpréteur **11**

3.1 Bytecodes 11

3.1.1 Calculs arithmétiques 11

3.1.2 Constantes 12

3.1.2.1 Constantes implicites 12

3.1.2.2 Constantes explicites 12

3.1.3 Variables locales et attributs 13

3.1.3.1 Variables locales 13

3.1.3.2 Variables attributs 14

3.1.4 Tableau (array) 14

3.1.4.1 Obtenir la taille d'un tableau 14

3.1.4.2 Obtenir la valeur d'un élément d'un tableau 15

3.1.4.3 Modifier la valeur d'un élément du tableau 15

3.1.4.4 Exceptions 15

3.1.5 Sauts conditionnels et non-conditionnels . . . 16
3.1.5.1 Comparaisons . . . 16
3.1.5.2 Sauts relatifs non-conditionnels . . . 16
3.1.5.3 Sauts conditionnels . . . 17
3.1.6 Invoke static et non-static . . . 17
3.1.6.1 Invoke static . . . 17
3.1.6.2 Invoke non-static . . . 17
3.1.7 Exceptions . . . 18
3.2 Syntaxe & utilisation . . . 18
3.2.1 XML : eXtensible Markup Langage . . . 18
3.2.1.1 Les principales règles . . . 18
3.2.1.2 Le parseur . . . 19
3.2.2 Attributs . . . 19
3.2.3 Modularité . . . 19
3.2.3.1 Configuration de la VM . . . 19
3.2.3.2 Dépendance des bytecodes . . . 20
3.2.4 Sécurité . . . 20
3.2.4.1 Vérification . . . 21
3.2.4.2 Arguments . . . 21
3.3 Générateur d'interpréteur . . . 22
3.3.1 Bytecodes simples . . . 22
3.3.2 Bytecode avec vérifications . . . 23
3.3.3 Bytecode fonction . . . 24

4 Optimisation de l'interpréteur 25
4.1 Plateforme matérielle . . . 25
4.2 Compilateur . . . 25
4.3 Optimisation . . . 26
4.3.1 Tests . . . 26
4.3.2 Comparaison . . . 26
4.3.3 Résultat . . . 27

5 Déroulement du stage 28
5.1 Déroulement du stage . . . 28
5.2 Environnement de travail . . . 29
5.3 Connaissances acquises . . . 29

Introduction

Dans le cadre de mon stage de fin d'année, j'ai travaillé pendant dix semaines au sein de l'équipe RD2P.

Ce stage, axé sur le domaine des systèmes embarquès, comportait de la programmation en C et Java.

J'ai été amené à travaillé sur le projet JITS. L'objectif principal du projet étant de faire fonctionner du Java dans des Petits Objets Portables et Sécurisés sans pour autant dégrader les services offerts par le langage.

Je vous propose dans un premier temps de découvrir le contexte où s'est déroulé mon stage. Puis on découvrira ce qu'apporte le projet JITS. Et enfin, je vous présenterai mon travail ainsi que le déroulement du stage.

Chapitre 1

Présentation du LIFL[1]

1.1 Introduction

Le Laboratoire d'Informatique Fondamentale de Lille (LIFL) est une unité de recherche de l'Université des Sciences et Technologies de Lille (Université de Lille-1). Le LIFL est associé au département Science et Technologies de l'Information et de Communication du CNRS.

Le LIFL, depuis sa création en 1983, a connu une forte croissance. En effet, on dénombre aujourd'hui près de 160 personnes travaillant au LIFL.

Le LIFL est également en liaison avec de nombreuses filières de formation en informatique dans lesquels interviennent les enseignants-chercheurs de l'unité :

- DUT Informatique
- Maîtrise d'Informatique
- IUP Miage et IUP Systèmes Informatiques
- DESS (Téléinformatique et Informatique Répartie, Intelligence Artificielle et Génie Logiciel, Multimédia et Internet pour le Commerce Electronique)
- Ecole d'ingénieur EUDIL
- Ecole Universitaire d'Ingénieurs de Lille - (Filières « Informatique, Mesure, Automatique » et « Génie Informatique et statistique »)
- Ecole d'ingénieur ENIC Ecole Nouvelle d'Ingénieur en Communication

Le LIFL dispose également d'une infrastructure informatique dédiée à la recherche et au développement. Son équipement comprend plus de 200 postes de travail (Sun, SGI, PC linux, PC windows, Macintosh) ainsi que des stations graphiques spécialisées (SGI, PC). Tous ces postes sont reliés en réseau et disposent des services communs offerts par les serveurs.

[1] Laboratoire d'Informatique Fondamentale de Lille

1.2 Axes et équipes

Les recherches menées au LIFL sont regroupées autour de trois axes comportant plusieurs thèmes :

- Une composante théorique autour du Calcul Formel, de la Bio-Informatique et des Spécifications (CBS).
- Une composante plus tournée vers les nouvelles applications et interfaces pour les réseaux : Coopération, Image et Mobilité (CIM) avec des spécificités autour des composants logiciels, de la carte à puce et des simulateurs médicaux.
- Une composante autour des environnements distribués et applications parralèles : Simulation, Optimisation , Calcul Scientifiques (SCOPE).

Ces trois thèmes ne sont pas totalement indépendants, les équipes sont plus ou moins liées entre elles.

Le déroulement du stage s'est effectué chez l'équipe RD2P. Cette équipe appartient à l'axe CIM.

1.3 L'équipe RD2P

L'équipe RD2P est le résulat d'un partenariat entre Gemplus Card Internationnal et Université de Lille I. Ses recherches sont principalement centrées sur la faisabilité des POPS (Petits Objets Portables et Sécurisés) et plus particulièrement sur les aspects « système » et « réseaux » qu'offre ces petits systèmes. On distingue trois principaux axes de recherche :

- Réseaux mobiles ;
- Protection de l'information ;
- Systèmes embarqués.

Parmi les POPS, on peut citer les cartes à microprocesseur, thème fondateur de l'équipe, les étiquettes électroniques ou encore les assistants personnels électroniques.

Chapitre 2

Java In The Small

2.1 Différentes versions et différentes VM

2.1.1 CDC

CDC[1] fournit une machine virtuelle ainsi que l'API de base. C'est une version de Java quasi-complète (par rapport au J2SE[2]). Elle se destine pour des machines « performantes » tels que les PDAs haut de gamme. Pour un système embarqué, elle nécessite une configuration robuste : un processeur 32 bits (plus 50 Mhz), entre de 2 MB et 16 MB de mémoire (ROM+RAM) et d'une bande passante d'au moins 56kb/s. Cette version du J2ME[3] possède sa propre machine virtuelle : la CVM[4] . La CVM possède les mêmes fonctionnalités qu'une JVM classique. Sa principale différence est sa taille réduite. Bien entendu, ce dispositif fonctionne au dessus du système d'exploitation hôte.

remarque Cette version de Java demande une machine trop performante pour pouvoir être utilisé sur une plateforme embarquée.

2.1.2 CLDC

CLDC[5] est une autre version du J2ME. Comme son nom l'indique, elle est destinée à fonctionner sur des systèmes limités (téléphones portables, PDAs, ...). CLDC est capable de fonctionner avec une petite configuration : processeur 16 ou 32bits, entre 160kB et 512KB de mémoire et une bande passante de 9600 bauds/s.

Pour pouvoir rendre ce dispositif fonctionnel sur des petits objets, Sun décrit une nouvelle spécification ainsi qu'une nouvelle machine virtuelle (la KVM[6]). Dans cette nouvelle spécification, on retrouve certaines contraintes :

- la suppression de certains types (entier 64bits, nombre à virgule flottante) ;

[1] Connected Device Configuration.
[2] Java 2 Standard Edition.
[3] Java 2 Micro Edition (CDC et CLDC).
[4] Compact Virtual Machine.
[5] Connected Limited Device Configuration.
[6] Kilobyte Virtual Machine.

- l'absence de JNI [7] : impossiblité d'écrire de nouvelles méthodes natives ;
- l'introspection (ou la réflexion) sur une classe ou un objet n'est plus disponible ;
- les groupes de threads et les daemons n'existent plus ;
- le manque de certaines exceptions.

remarque Le principal inconvénient est le fait de mettre une JVM au dessus d'un OS (redondance de fonctionnalités) ne laisse plus de place pour l'application à excuter.

2.1.3 JavaCard

JavaCard, à la différence de CLDC, n'utilise pas de système d'exploitation. JavaCard est une autre spécification de Sun pour les POPS fortement limités (carte à microprocesseur). Elle dispose aussi d'une nouvelle machine virtuelle : la CardVM. La CardVM est capable de fonctionner sur des processeurs 8 bits disposant de faible ressource.

Celle-ci offre, en plus des restrictions de CLDC, de nouvelles contraintes :

- les types codés sur 8 bits et 16 bits sont les seuls disponibles ;
- le support du multithreading n'existe plus.

remarque L'inconvénient majeur est la ré-écriture de son application pour JavaCard.

2.2 JITS : Java In The Small

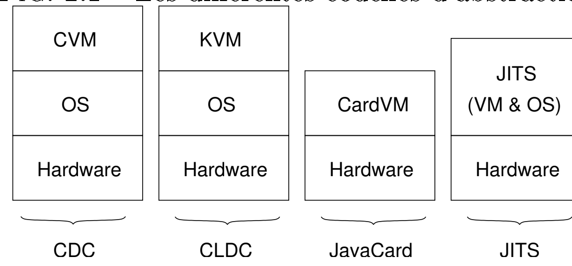
JITS est un projet de recherche dont le but est de simplifier le développement en Java pour plate-forme embarquée. A la différence des implémentations actuelles, JITS se propose d'offrir les mêmes fonctionnalités que la version classique du language (J2SE) sans dégradation importante des services offert par la VM.

JITS n'est pas vraiment une JVM dans la mesure où il assure aussi le rôle du système d'exploitation hôte (cf. figure 2.1). Cela enlève tous les problèmes de redondance de fonctionnalités voir même d'inutilé de certains services offert par le système d'exploitation. Pour minimiser la dépendance matérielle de ce JVM-OS, le développement de JITS se fait majoritèrement en Java. Il dispose des mêmes fonctionnalités qu'une VM classique :

- une API entièrement compatible avec la version classique du language ;
- un interpréteur de bytecodes performant ;
- un gestionnaire de mémoire[8]s'appuyant sur l'outil *preverify* ;
- support du multithreading :*Green Threading* (ce sont des threads Java indépendant du matériel et de l'OS) ;
- un ClassLoader entièrement écrit en Java. Ce qui est une première dans le domaine des JVM accessible au publique.

[7] Java Native Interface.
[8] Garbage Collector.

Fig. 2.1 – Les différentes couches d'abstraction



En plus de ces fonctionnalités, JITS dispose d'un ensemble d'outils dont le *romizer.* Cet outil permet de résoudre les dépendances entre classes java et leurs chargements avant la création de la ROM (JVM+OS). De ce fait, le démarrage de la VM est nettement accèleré.

Chapitre 3

Générateur d'interpréteur

3.1 Bytecodes

Un bytecode, comme son nom l'indique, est une instruction codée sur un seul octet. Cette instruction élémentaire fait référence à une suite d'opérations (dépilement/empilement, calcul arithmétique, accès à un tableau, ...). Il se peut qu'un bytecode ait besoin de un ou plusieurs paramètres. De manière générale, ces paramètres se trouvent sur le sommet de la pile ou dans la « heap »[1]. Mais il se peut aussi que ces paramètres se trouvent dans le flux d'octets suivant ce bytecode.

3.1.1 Calculs arithmétiques

La notation des bytecodes arithmétiques est écrite de la sorte. La première lettre fait référence au type (int, long, float, double). Et la suite de lettres correspond à l'opération arithmétique (add pour addition, sub pour soustraction , mul pour multiplication , div pour division , rem pour reste). Cette opération arithmétique peut être aussi bien binaire (addition) que unaire (négation) .

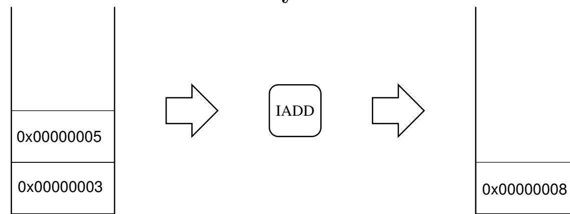
Par exemple :

- *iadd* correspond à l'addition de deux nombres entiers ;
- *fmul* correspond à la multiplication de deux nombres à virgule flottante.

Pour les calculs arithmétiques, le calcul se fait en plusieurs étapes. La première étape consiste à dépiler la ou les opérandes dont on a besoin. Puis la machine virtuelle effectue son calcul. Et enfin, le résultat est empilé (cf. figure 3.1).

[1] cf. 2.1 on the preceding page.

Fig. 3.1 – bytecode : *iadd*



Pour tous les calculs, le principe reste le même. Avant d'effectuer son calcul, la machine virtuelle a besoin de dépiler les opérandes dont elle a besoin. Puis, une fois le calcul terminé, le résultat est empilé.

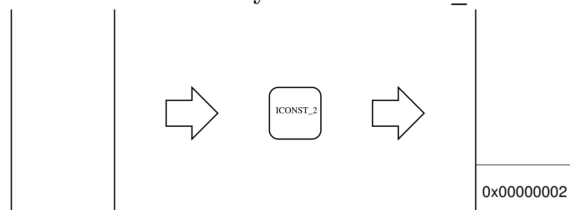
3.1.2 Constantes

3.1.2.1 Constantes implicites

Ce sont les bytecodes les plus simples. Il ne font qu'empiler une valeur sur la pile et ne prennent aucun paramètres (cf. figure 3.2).

Par exemple : iconst_0, iconst_1, icont_2 fconst_0, fconst_1, dconst_0, dconst_1

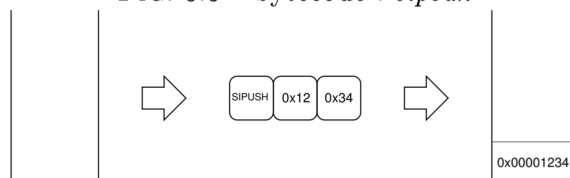
Fig. 3.2 – bytecode : *iconst_2*



3.1.2.2 Constantes explicites

Par exemple *bipush* et *sipush* en sont le parfait exemple. Ces bytecodes se contente juste d'empiler une valeur entière. *bipush* empile un byte (compris entre -128 et +127), tandis que *sipush* (cf. figure 3.3) empile un short (compris entre -32768 et 32767). Cette valeur se trouve juste après le bytecode.

Fig. 3.3 – bytecode : *sipsuh*



3.1.3 Variables locales et attributs

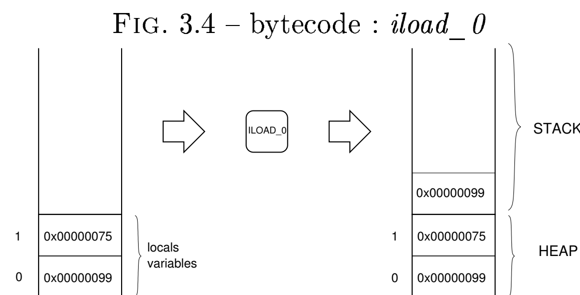
Par définition les variables locales sont des variables existantes seulement à l'intérieur d'une méthode (les paramètres d'une méthode sont considérés comme des variables locales à parts entières). Les variables locales se situent dans une zone mémoire appelée la « heap ». Contrairement à la pile, l'accès aux éléments la « heap » se fait sans empilement ou dépilement. En effet, la « heap » être representée par un tableau. Et comme pour tout tableau, l'accès se fait à l'aide d'un indice.

3.1.3.1 Variables locales

1.1. Obtenir la valeur d'une variable locale

Pour connaître la valeur d'une variable locale, l'indice de position est obligatoire. Pour les bytecodes constants de type *xload_* $<n>$ ou $<n>$ est un indice, aucun paramètre n'est requit (cf. figure 3.4). Par contre pour les bytecodes variables de type *xload*, l'octet suivant nous indique l'indice. Dans les deux cas, la valeur de la variable souhaitée est empilée. Exemple : iload_0,iload_1,iload_2,iload_3, fload_0,.. ... iload, lload,fload, dload.

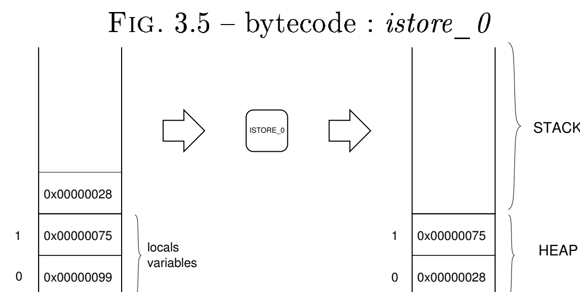
FIG. 3.4 – bytecode : *iload_0*



1.2. Modifier la valeur d'une variable locale

Pour modifier le contenu d'une variable locale, l'indice de position reste toujours obligatoire. On retrouve deux types de bytecodes *xstore_* $<n>$ et xstore fonctionnant de la même manière que *xload_* $<n>$ et *xload*. Bien entendu, la nouvelle valeur doit être préalablement empilée (cf. figure 3.5). Ici, aucun résultat n'est empilé. Exemple : istore_0, istore_1, fstore_0, ... istore, lstore, fstore, dstore.

FIG. 3.5 – bytecode : *istore_0*

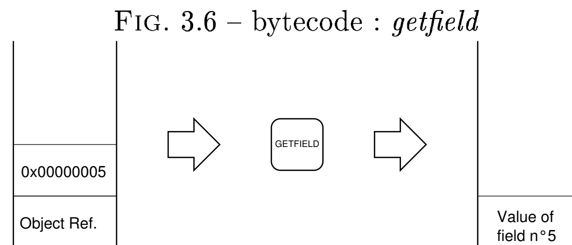


3.1.3.2 Variables attributs

2.1. Obtenir la valeur d'un attribut

Pour obtenir la valeur de l'attribut d'un objet, il suffit d'empiler la réference de l'objet souhaité et l'indice de cet attribut (cf. figure 3.6). Le résultat peut être codé sur 32 bits ou 64 bits suivant le type de cet attribut. Exemple : getfield.

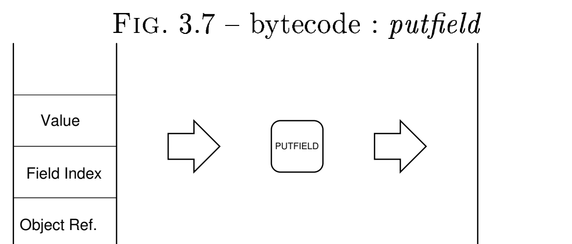
FIG. 3.6 – bytecode : *getfield*



2.2. Modifier la valeur d'un attribut

Pour modifier la valeur de l'attribut d'un objet, la réference de l'objet souhaité, l'indice de cet attribut et la valeur de cet attribut doivent être empilées préalablement (cf. figure 3.7). Bien sûr, aucun résultat n'est empilé. Exemple : putfield.

FIG. 3.7 – bytecode : *putfield*



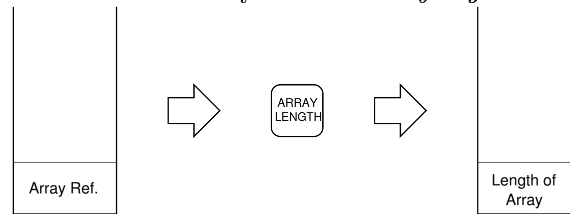
3.1.4 Tableau (array)

Pour les tableaux, le fonctionnement est le même que pour les opérations arithmétiques. Sauf que dans la pile, on ne retrouve pas seulement des entiers mais aussi la référence d'un tableau. Cette référence, codée sur 32 bits, est liée à un objet java (array). L'une des particularités du langage java, est que les tableaux sont des objets à parts entiers. Et comme tout objet, un tableau possède des attributs (length par exemple), des méthodes (public String toString(), int hashcode() , ...) et un constructeur.

3.1.4.1 Obtenir la taille d'un tableau

Pour obtenir la taille d'un tableau, la réference du tableau doit être empilée préalablement (cf. figure 3.8). Le résultat (la taille) est empilé. Ce résultat peut être égal à zéro. Exemple : arraylength.

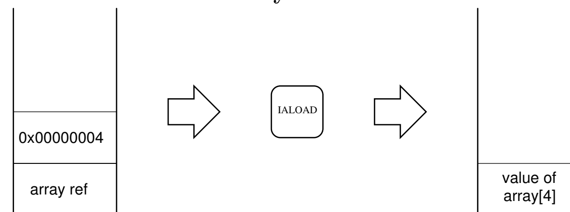
FIG. 3.8 – bytecode : *arraylength*



3.1.4.2 Obtenir la valeur d'un élément d'un tableau

Pour obtenir la valeur d'un élément d'un tableau, il suffit d'empiler la référence du tableau et l'index de l'élément (compris entre 0 et length-1) (cf. figure 3.9). Le résultat fournit se trouve sur le sommet de la pile. Il peut être codé sur 32 ou 64 bits suivant le type des éléments du tableau. Exemple : iaload, faload, laload, daload, baload, saload, caload, aaload.

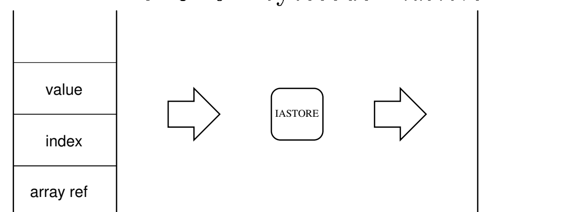
FIG. 3.9 – bytecode : *iaload*



3.1.4.3 Modifier la valeur d'un élément du tableau

Pour modifier la valeur d'un élément du tableau, il suffit d'empiler la référence du tableau, l'index de l'élément et la nouvelle valeur (cf. figure 3.10). Bien entendu, il n'y a pas de résultat sur le sommet de pile. Exemple : iastore, fastore, lastore, dastore, bastore, sastore, castore, aastore.

FIG. 3.10 – bytecode : *iastore*

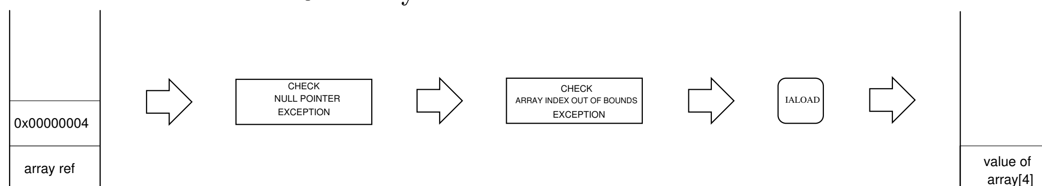


3.1.4.4 Exceptions

Pour des questions de sécurité, un certain nombre de tests sont fait sur la référence du tableau ainsi que sur l'index utilisé. Si une erreur survient, une exception associée à cette erreur

est obligatoirement lancée (cf. figure 3.11). Par exemple, l'exception *NullPointerException* est lancée si la référence du tableau est égale à *null*. Par contre, si l'index est invalide une exception *ArrayOutOfBoundsException* est lancée.

Fig. 3.11 – bytecode : *iaload* avec vérifications

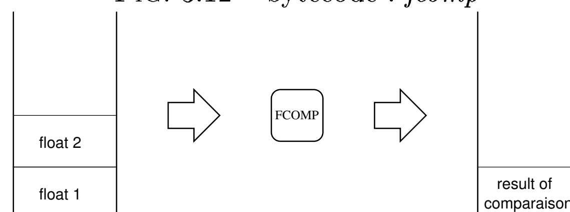


3.1.5 Sauts conditionnels et non-conditionnels

3.1.5.1 Comparaisons

La comparaison s'effectue sur les deux premiers éléments du sommet de pile. Le résultat de la comparaison est empilé (cf. figure 3.12). Si les deux éléments sont égaux, le résultat empilé est zéro. Sinon, le résultat empilé est 1 ou -1 si l'un des éléments est supérieur ou inférieur à l'autre. Ce résultat est utilisé pour le saut conditionnel. Exemple : fcomp, dcomp, lcomp.

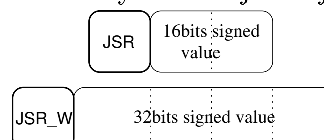
Fig. 3.12 – bytecode : *fcomp*



3.1.5.2 Sauts relatifs non-conditionnels

A la différence d'un saut absolu, un saut relatif incrémente la position du pointeur d'instruction (pc[2]). Si le bytecode est *jsr* alors le pointeur d'instruction est incrémenté de la valeur du mot (2 octets) suivant ce bytecode (cf. figure 3.13). Par contre, si le bytecode est *jsr_w*, ce mot est codé sur 4 octets.

Fig. 3.13 – bytecode : *jsr* & *jsr_w*



[2]Program Counter.

3.1.5.3 Sauts conditionnels

Le saut conditionnel dépend du sommet de pile. Si le sommet de pile répond à la condition demandé alors un saut relatif peut être effectué. Ce saut relatif est éffectué avec les deux octets suivant ce bytecode. Parmi les conditions, on distingue l'égalité, la non-égalité, la supériorité ou l'infériorité. Voici, les bytecodes possibles :

- ifeq, sommet de pile égale à zéro ;
- ifneq,sommet de pile différent de zéro ;
- ifgt, sommet de pile supérieur à zéro ;
- iflt, sommet de pile inérieur à zéro.

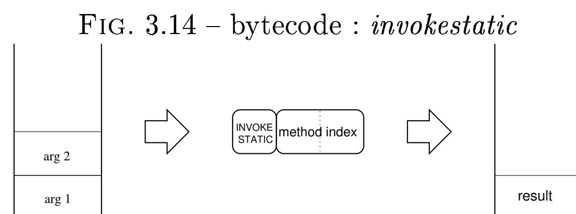
3.1.6 Invoke static et non-static

Il existe plusieurs type de invoke (*invokeinteface*, *invokespecial*, *invokevirtual*, *invokestatic*). On distingue deux catégories : les static et les non-static. En effet, hormis le bytecode *invokestatic*, les autres ont tous le même fonctionnement (par rapport à la pile). Dans tout les cas, une frame est construite (empilé). Une frame n'est rien d'autre que l'empilement d'éléments (paramètres de la méthode, variables locales, ...) suivit de l'empilement de nombre d'éléments. On peut aussi considéré la frame comme une mini-pile. De plus, la frame s'avère très utile pour la gestion des exceptions .

3.1.6.1 Invoke static

L'invoke static s'applique sur une classe. Mais on n'a pas besoin d'empiler la classe associée à cette méthode statique (cf. figure 3.14). Ensuite, une fois ces paramètres empilés, l'appel de la méthode est possible. Les paramètres sont depilés automatiquement par la méthode appelée. Et enfin, le résultat est empilé si c'est une méthode non-void.

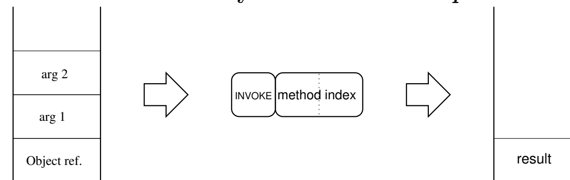
FIG. 3.14 – bytecode : *invokestatic*



3.1.6.2 Invoke non-static

L'invoke non-static s'applique toujours sur un objet. Il suffit donc d'empiler l'objet ainsi que ces paramètres avant d'appeler la méthode non-static (cf. figure 3.15). Bien entendu, un résultat est empilé si c'est une méthode non-void.

FIG. 3.15 – bytecode : *invokespecial*



3.1.7 Exceptions

Les exceptions en Java sont une manière élégante de traiter les erreurs. Elles sont très liées aux méthodes. En effet, chaque méthode possède une table des exceptions. Dans cette table, chaque bloque try/catch y est représenté. Lors qu'une exception est lancée, chaque frame est dépilée jusqu'a qu'une méthode « attrape » l'exception. Si aucune méthode attrape cette exception, c'est alors la machine virtuelle de la traiter.

3.2 Syntaxe & utilisation

3.2.1 XML : eXtensible Markup Langage

Dérivé d'un language nommé SGML (Standard Generalized Markup Language), le XML a su s'imposer dans de multiple domaines. Outre sa capacité de modéliser et stocker des données, ce standard offre une indépendance vis à vis de la plateforme utilisée. Cette caractéristique lui donne une grande interporabilité entre différénts systemes.De plus, dans la mesure ou il y n'a pas de tags prédéfinis, ce métalangage permet de définir de nouvelles balises sans aucun problème.

Pour le différencier des autres langages dérivés du SGML, un fichier XML contient un DTD. (Document Type Definition) pouvant indiquer quelques informations comme le numéro de version.

3.2.1.1 Les principales règles

Pour qu'un document XML soit valide, un certain nombre de règles doit être respecté. Pour pouvoir être chargé, le document doit avoir une syntaxe correcte. Voici les règles importantes à respecter :

- le document doit contenir au moins une balise;
- chaque balise d'ouverture doit possèder une balise de fermeture (<tag>..</tag>) ou peut être abrégé (<tag/>) ;
- les valeurs des attributs doivent être entre guillemés;
- une distinction entre majuscule et minuscule est faite.

3.2.1.2 Le parseur

Pour charger un document XML, il existe deux méthodes fournise avec l'API java : SAX (Simple API for XML) et DOM (Document Object Model). C'est la méthode DOM qui a été utilisée. Elle est l'oeuvre des recommandations du W3C.

Pour manipuler des données, une représentation en un arbre est généré. On dispose de toutes primitives de base pour parcourir cet arbre.

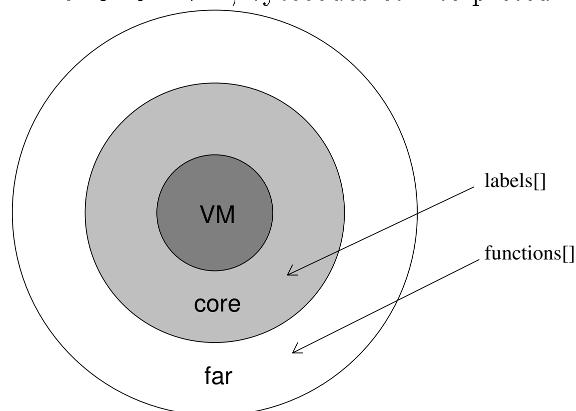
3.2.2 Attributs

Pour être modéliser, un bytecode dispose d'attributs minimales :

- bc : un numéro compris entre 0 et 255 (inclus) ;
- name : un nom indiquant son fonctionnement ;
- pop/push : Ces attributs nous indiquent le fonctionnement du bytecode au niveau de la pile. Les valeurs de ces attributs sont le nombre de mots de 32 bits empilés ou dépilés ;
- inc_pc : Cet attribut nous indique le nombre de octets suivant ce bytecode. Ces octets sont utilisés comme paramètres. Ces paramètres peuvent être l'indice d'une variable locale, l'indice de l'attribut d'un objet, l'indice d'une méthode d'un objet ou une valeur constante directe (codée sur 8 ou 16 bits) ;
- inCore : La valeur de l'attribut prend "1" ou "0". Si il est activé (égal à 1) , le bytecode est interprété rapidement mais ne peut être interrompu. Sinon, il devient une fonction. Et l'interpréteur doit alors accéder à la table des fonctions (cf. figure 3.16).

  Par exemple, iadd s'écrit de la sorte :
  <bytecode bc="96" name="iadd" pop="2" push="1" inc_pc="0" inCore="1"/>

Fig. 3.16 – VM, bytecodes et interpréteur



3.2.3 Modularité

3.2.3.1 Configuration de la VM

Les principales fonctionnalités de la VM sont le support des nombres à virgules flottantes (float et double) et le support des types codés 64 bits (long et double).

Une configuration avec ces options (support des flottants et des entiers codés sur 64 bits) donnerait :

```
<vmconfig>
<vmprop name="vmfpu"/>
<vmprop name="vm64bits"/>
</vmconfig>
```

3.2.3.2 Dépendance des bytecodes

Certains bytecodes dépendent de la configuration de la VM. De ce fait, si le support d'une fonctionnalité n'est plus souhaité, toutes les implémentations qui en découle ne sont plus générées. Cela permet d'offrir une VM sur mesure à l'application developpée ainsi qu'un gain de place considérable.

Par exemple, si on supprime le support des types codés sur 64 bits alors l'arithmétique et les opérations sur les tableaux d'entiers long ne sont plus supportés.

La notations d'un bytecodes dépendant du support 64bits s'écrit alors :

```
<bytecode bc="ladd" pop="4" push="2" inCore="1" >
<require><vmprop name="vm64bits"/></require>
</bytecode>
```

Si le bytecode dépend deux propriétés, il se note :

```
<bytecode bc="dadd" pop="4" push="2" inCore="1" >
<require><vmprop name="vmfpu"/>
</require> <require><vmprop name="vm64bits"/>
</require> </bytecode>
```

Et si le bytecode ne dépend d'aucune propriété, on peut l'écrire :

```
<bytecode bc="dadd" pop="4" push="2" inCore="1" >
</bytecode>
```

ou

```
<bytecode bc="dadd" pop="4" push="2" inCore="1" />
```

3.2.4 Sécurité

Le grand succès de Java est surtout la sécurité qu'il apporte. En effet, en Java, l'écriture dans une zone mémoire autre que celle autorisé est impossible. En cas d'erreur, une exception est donc lancée.

3.2.4.1 Vérification

De manière générale, une vérification est faite sur certains paramètres du bytecode. Si cette vérification échoue alors une exception est lancée. Le nom de l'exception nous permet de connaître le type d'erreur.
Au niveau du bytecode, quelques vérifications sont effectuées. Les exceptions qui peuvent être lancées sont :

- NullPointerException ;
- ArrayIndexOutOfBoundsException ;
- NegativeArraySizeException ;
- ArithmeticException ;
- ClassCastException.

Les vérifications sont effectués par les macros suivantes :

- CHECK_NULLPOINTEREXCEPTION(REF) ;
- CHECK_ARRAYINDEXOUTOFBOUNDSEXCEPTION(REF,INDEX) ;
- CHECK_NEGATIVEARRAYSIZEEXCEPTION(SIZE) ;
- CHECK_ARITHMETICEXCEPTION(NUM) ;
- CHECK_CLASSCASTEXCEPTION(REF,CLAZZ).

3.2.4.2 Arguments

Pour le passage de paramètre, il faut spécifié où se trouve le paramètre (pile, constante directe, constant indirecte), le type du paramètre (byte, short, int , long, float, double, Object, Array) et son indice.

Par exemple, pour utiliser :

- le premier nombre à virgule flottante de la pile :

<arg><stack type="float" index="1"/></arg>

- le second octet constant direct :

<arg><cst type="byte" index="2" /></arg>

- le premier mot codé sur 16 bits faisant référence au constante pool :

<arg><cstPl type="short" index="1"/></arg>

Le bytecode permettant d'obtenir une valeur provenant un tableau d'entier s'écrit avec tous les vérifications possibles :

```
<bytecode bc="46" name="iaload" pop="2" push="1" inCore="1">

<exception name="NullPointerException">
<arg><stack type="Array" index="2"/></arg>
</exception>

<exception name="ArrayIndexOutOfBoundsException">
<arg><stack type="Array" index="2"/></arg>
<arg><stack type="int" index="1"/></arg>
</exception>
</bytecode>
```

3.3 Générateur d'interpréteur

A partir des informations sur les bytecodes et sur la configuration de la VM, le générateur d'interpréteur peut creer toutes la boucle principale d'interprétation. Il doit générer le tableau de labels (ou le tableau de branchements). Le générateur s'appuit sur des macros (DEF_BC(xxx) est une macro pour designer un label, TOS étant le sommet de pile, DEF_CONTINUE fait un saut au bytecode suivant, DEF_END fait aussi un saut vers le bytecode suivant mais permet aussi de changer de thread). Le comportement des macros doit être écrit.

Ici, on présentera quelques exemples sur la génération du fonctionnement des bytecodes.

3.3.1 Bytecodes simples

Ces deux exemples montrent comment utiliser la syntaxe XML pour générer le fonctionnement des bytecodes.

Le bytecode *iadd* qui s'est en XML :

```
<bytecode bc="96" name="iadd" pop="2" push="1" inc_pc="0" inCore="1"/>
```

sera généré :

```
DEF_BC(IADD)
 BEGIN
  DEF_BODY_IADD
  TOS –;
 END
DEF_END
```

Le bytecode *sipush* servant à empiler une valeur :

```
<bytecode bc="17" name="sipush" pop="0" push="1" inc_pc="2" inCore="1"/>
```

donnera :

```
DEF_BC(IADD)
 BEGIN
  DEF_BODY_IADD
  PC += 2;
 END
DEF_CONTINUE
```

3.3.2 Bytecode avec vérifications

Cet exemple sert juste à montrer comment sont généré les vérifications sur certains bytecodes. Ces vérifications sont faite par des macros.

Le bytecode *iaload* servant à obtenir la valeur d'un élément d'un tableau d'entier s'écrit avec toutes les vérifications possibles :

```
<bytecode bc="46" name="iaload" pop="2" push="1" inCore="1">

<exception name="NullPointerException">
<arg><stack type="Array" index="2"/></arg>
</exception>

<exception name="ArrayIndexOutOfBoundsException">
<arg><stack type="Array" index="2"/></arg>
<arg><stack type="int" index="1"/></arg>
</exception>

</bytecode>
```

se généré de la sorte :

```
DEF_BC(IALOAD)
 BEGIN
  CHECK_NULLPOINTEREXCEPTION(ATOS(2))
  CHECK_ARRAYINDEXOUTOFBOUNDSEXCEPTION(ATOS(2),ITOS(1))
```

```
 DEF_BODY_IALOAD
 TOS --;
END
DEF_END
```

3.3.3 Bytecode fonction

Un bytecode de type fonction (inCore="0") n'est pas généré de la même manière que les autres bytecodes. En effet, un bytecode de ce type doit passer par l'intermédiaire d'une table des fonctions. Un bytecode est mis en fonction lorsque celui-ci est complexe.

Le bytecode *new* servant à creer un objet se notant :

```
<bytecode bc=187 name="new" pop="0" push="1" inc_pc="2" inCore="0"/>
```

sera alors dans la table des fonctions et donnera :

```
lablel_FUNCTIONS :
 SAVE_CONTEXT
 functions_table [ bc - FIRST_FUNCTION_INDEX ]()
 RESTORE_CONTEXT
DEF_END
```

La définition du bytecode devient une fonction et s'écrit alors :

```
void new_func (void )
{
 DEF_BODY_NEW
 PC +=2;
}
```

Chapitre 4

Optimisation de l'interpréteur

Hélas, à mon arrivé la VM de JITS était relativement lente par rapport à celle de Sun. De ce fait, une partie de mon travail fut donc d'optimiser l'interpréteur de bytecode. Ceci était un travail long et fastidieu dans la mesure où l'interpréteur généré devait être optimisé à la fois pour une architecture Intel ia32 (pour la comparaison avec la VM de Sun) et une architecture embarquée de type RISC[1] (ARM7).

4.1 Plateforme matérielle

La principale caractéristique des machines embarquées est surtout l'hétérogéinité du matériel. Néanmoins, la tendance actuelle est aux processeurs ARM. En effet, le faible coût de revient et le bon rapport puissance / consommation d'énergie lui donne l'avantage face à leurs concurrents.

A la différence d'un CISC[2], toutes les instructions sont codées sur une taille fixe. Un processeur ARM dispose souvent de deux modes : le mode ARM (chaque instruction est codée sur 32 bits) et le mode THUMB (chaque instruction est codée sur 16 bits). Une autre caractéristique des processeurs de type RISC est aussi le nombre important de registres.

4.2 Compilateur

Le compilateur utilisé par JITS est GCC[3]. Ce compilateur multiplateforme permet de compiler aussi bien pour une architecteure x86 que pour une architecture ARM.

Concernant l'architecture x86, le compilateur ICC[4] du celèbre constructeur n'a pas pu être testé. Bien que performant, il a posé des problèmes de compatibilité. En effet, les tableaux de labels ne sont pas acceptés. De ce fait, il n'apparaît pas dans les tests de performances.

[1]Reduced Instruction Set Computer.

[2]Complex Intructions Set Computer.

[3]Gnu Collection Compiler

[4]Intel Cross Compiler

4.3 Optimisation

Les tests ont été éffectués sur un Pentium III 500 Mhz avec désactivation du « JIT compiler ».

4.3.1 Tests

Une série de trois tests sur des boucles « for » de 1000000 d'itérations ont été faite. Ces boucles mettent en jeu plusieurs bytecodes (incrémentation, comparaison, saut conditionnel, ...).

Voici les tests :

Algorithm 1 TestEmptyLoop

```
for ( int i = 0 ; i < 1000000 ; i++)
;
```

Algorithm 2 TestEmptyDoubleLoop

```
for ( int i = 0 ; i < 1000 ; i++) {
 for ( int j = 0 ; j < 1000 ; j++)
;
}
```

Algorithm 3 TestEmptyTripleLoop

```
for ( int i = 0 ; i < 100 ; i++) {
 for ( int j = 0 ; j < 100 ; j++) {
  for ( int k = 0 ; k < 100 ; k++)
   ;
 }
}
```

4.3.2 Comparaison

Les premiers tests ont été effectué en utilisant la version 3.2 de Gcc avec l'option -O3. Le gros problème était l'accès dans la table des labels au bytecode suivant. En effet, l'assembleur généré pouvait nettement amélioré. Les tests suivant ont été fait sur la version 3.4 toujours avec l'option -O3. L'accès à la table des labels s'est nettement amélioré. Puis, pour palier à ce problème, une optimisation manuelle s'est faite sur le code source de l'assembleur généré. L'accès à la table des labels à été réécrite de manière optimale en assembleur (cf. Résultat).

4.3.3 Résultat

TAB. 4.1 – Performances

	JITS (gcc 3.2)	JITS (gcc 3.4)	JITS[5] (optimisation)	HotSpot[6] (interpréteur)
TestEmptyLoop	341ms	188ms	70ms	87ms
TestEmptyDoubleLoop	340ms	190ms	67ms	62ms
TestEmptyTripleLoop	340ms	181ms	63ms	93ms

Les résultats montrent que les optimisations du compilateur ne suffisent pas. Néanmoins, ces optimisations manuelles sont spécifiques à une plate-forme. Elles posent des problèmes de portablité.

Chapitre 5

Déroulement du stage

5.1 Déroulement du stage

Le but de mon travail consistait à écrire un générateur d'interpréteur de bytecode. Par définition, l'outil créé se veut hautement paramétrable.

A mon arrivé, Alexandre Courbot me montra le fonctionnement de JITS ainsi que son organisation interne. En effet, la bonne organisation du projet JITS en sous-arborescence m'a facilité la tâche. Ensuite, il me conseilla de me documenter sur le site de Sun, et plus particulierement la spécification de la VM communément appelée « la Spéc ». Ainsi ce fut pour moi, l'occasion de découvrir le fonctionnement d'une machine à pile. Du bytecode le plus simple au plus complexe, mon premier jour fut consacré à la réunion de toutes les informations néccessaires.

Puis lors d'une réunion avec Gilles Grimaud et Alexandre Courbot, on me décriva le travail à effectuer ainsi que le développement actuel de JITS.

Etant donné que plusieurs personnes travaillent en continue sur le projet JITS, la familiarisation avec l'outil CVS fut indispensable. De plus, comme la documentation et le développement de JITS étaient anglophones, cela me permettait d'enrichir mon anglais.

Enfin, pour développer le générateur d'interpréteur, Alexandre Courbot me conseilla d'utiliser le format XML pour modéliser le fonctionnement des bytecodes. L'API java et sa documentation me permit de charger des fichiers au format XML contenant aussi bien la description des bytecodes que la configuration de la VM.

Bien entendu, au fur et à mesure que mon travail avancé, quelques informations s'ajouter à la modélisation des bytecodes. Au début, je commençais par générer les tableau de labels et de fonctions, puis le comportement des bytecodes devenait de plus en plus précis.

5.2 Environnement de travail

Le développement s'est fait entièrement sous linux (distribution Mandrake). Les principaux outils utilisés sont :

- Gnu CC : le compilateur multi-plateforme ;
- gdb : l'outil de débuggage ;
- Emacs : l'éditeur ;
- make : outil de compilation ;
- CVS : controleur de versions ;
- un émulateur pour effectuer des tests sur plate-forme ARM.

Pour me documenter, mes sources d'information proviennent surtout du site de Sun.

5.3 Connaissances acquises

Ce stage m'a permit d'apprendre le XML. J'ai pu aussi me familiariser avec des outils tels que CVS.

Bien entendu, j'ai pu apprendre davantage sur le plan technique. Mais les principales connaissances acquises se situent surtout au niveau de la méthodologie. En effet, face au nombre important de bytecodes (plus de 200), une rigueur s'est vite imposée. Une programmation rigoureusement orientée objet et une documentation de mon travail étaient plus que souhaitable. J'ai pu aussi apprendre davantage sur le développement entièrement anglophone et normalisé donnant un aspect plus professionnel à l'application.

Mais ce que j'ai apprécié le plus est l'organisation du projet JITS. En effet, l'arborescence de ce projet a bien été pensée. Chaque répertoire correspond à une fonctionnalité de la VM ou à un outil. On distingue chaque partie du projet. Parmi ces sous-partie, on a : api, tools, core, engine, omem, romizer, ... De plus la venue de l'utilisation du XML ne fait qu'accroître son organisation.

Conclusion

Ce stage m'a permis de découvrir un nouveau domaine informatique : l'embarqué. Cette approche de l'informatique donne lieu à de nombreuses contraintes.

Quelques nouveautés ont pu être apportées à un projet commun : JITS. Etant donné que plusieurs personnes travaillent sur ce projet de recherche, le respect de quelques règles de normalisation s'est vite imposé.

Concernant les connaissances acquises, ce stage m'a aussi permis d'approfondir ce qui a été enseigné à l'IUT. De plus, mon anglais écrit s'est amélioré grâce à la documentation de mes réalisations ainsi que mon anglais oral grâce à la venu de nouveaux stagiaires anglophones.

En outre, il m'a été agréable de constater que mes travaux seront utilisés. En effet, des nouveautés seront ajoutées sur la base qu'il m'a été demandé d'établir.

Glossaire

Bytecode instruction élémentaire interprétée par la machine virtuelle

CISC Complex Instruction Set Computer. Type d'achitecture disposant d'un grand nombre d'instructions, un faible nombre de registres. Exemple : Intel i386, Pentium.

CPU Central Processing Unit. C'est la partie du processeur qui traite, décode et exécute les instructions

FPU Floating Point Unit. Co-processeur arithmétique. Ce module s'occupe de tous les calculs à virgules flottante déchargeant ainsi le CPU de cette tâche.

Frame Environment d'excution d'une méthode

Instruction pointer Registre du processeur contenant l'adresse de la prochaine l'instruction à exécuter

JITS Java In The Small

JVM Java Virtual Machine. Machine Virtuelle Java

POPS Petits Objets Portables et Sécurisés

program counter : cf. instruction pointer

RD2P Recherche et Développement Dossier Portable

RISC Reduced Instruction Set Computer. Type d'architecture disposant d'un faible nombres d'instructions codées sur une taille fixe, d'un grand nombre de registre. Exemple : ARM7, PowerPC

SGML Un standard internationnal (ISO 8879) qui spécifie les règles de création des langages indépendants de la plate-forme, basés sur du « markup » ou des balises.

XML eXtensible Makup Language. Méta-langage extensible dérivé du SGML permettant de structurer des données.

Bibliographie

[1] G.Bizzotto. Java in the small.

[2] T. Lindholm and F.Yellin. The Java Virtual Machine Specification. ISBN 0-201-63452-X

[3] B. Kernighan and D.Richie. C Language.

[4] CDC, CLDC and JavaCard specifications. http ://java.sun.com/products/

[5] JITS. http ://www.lifl.fr/rd2p/jits