UPMC

Master informatique 2 – STL
NI503 – CONCEPTION DE LANGAGES
Notes I

Basile STARYNKEVITCH, http://starynkevitch.net/Basile/ *
travaillant au CEA, LIST sur **gcc-melt.org**
reprenant les notes du cours de
Pascal MANOURY, http://www.pps.univ-paris-diderot.fr/~eleph/

2013-2014

courriel : **basile@starynkevitch.net** *et* **basile.starynkevitch@cea.fr**

Labor omnia vincit

Virgile

A propos de ce cours

Attention : les *hyperliens* indiqués font partie du cours et *devraient être suivis et consultés*.

Note finale : 50% projets informatiques + 50% examen final (documents papiers autorisés, dispositifs informatiques [smartphone, tablette, ordinateur portable, etc....] interdits).

Votre code source doit être :

— sous **licence libre**, préférentiellement GPLv3+ ou CeCILL v2.1
— versionné, par exemple via `github` ou `gitorious`, voir `git`
— à **un ou deux auteurs** (travail en binôme [1]), clairement **identifiés par leur prénom et nom** et leur adresse de courriel et la mention `student at Université Pierre et Marie Curie, Paris, France` [2]
— me transmettre par messagerie avant le cours suivant votre travail.

*Toutes les notes de cours sont disponible sur mon site web.

1. Ceux qui auront le courage de travailler seuls pourront avoir un léger bonus.
2. Votre futur employeur verra votre code !

— langages de programmation acceptés (sans préférence) : Ocaml-4, C++11, C11, Rust, Go, Scheme et l'excellent SICP, Common Lisp, Clojure, Scala, Haskell, MELT etc... [3] ; Java n'est pas recommandé...
Il vous faut **apprendre *plusieurs* langages de programmation**, et savoir utiliser leur debogueur (comme `gdb`, `ocamldebug`, ...).
— codé en **anglais** (commentaires, noms de variables, documentation)
— identifier *clairement* le code qui n'est pas le vôtre - **éviter le plagiat** !
— exécutable (et compilable avec par exemple GNU make) sur **GNU/LINUX**/DEBIAN/SID/x86-64 en *ligne de commande* (je ne dois pas utiliser d'IDE comme eclipse pour compiler et tester votre code).

Objectifs de ce cours :
— Vous familiariser avec les notions de **sémantique** (et pragmatique !) des *langages de programmation*, facilitant ainsi l'apprentissage futur d'autres langages de programmations ;
— Vous donner les bases pour implémenter (ou adapter, ou étendre) un langage de programmation (ou de script) ou un "domain specific language"

le logiciel libre

Les quatre libertés `http://www.gnu.org/philosophy/free-sw.html`

0 la liberté d'exécuter le programme, pour tous les usages
1 la liberté d'étudier le fonctionnement du programme et de le modifier (accès au code source nécessaire)
2 la liberté de redistribuer (notamment sous forme source) le programme, donc d'aider son prochain
3 la liberté de distribuer aux autres des copies modifiées (code source)

Lire les différentes licences libres (les plus usuelles) : GPLv3 (obligation de publier les améliorations redistribuées sous la même licence), LGPLv3, Affero-GPL, MIT, BSD... Importance de la communauté !

Ne jamais inventer sa propre licence.

Qu'est-ce que le code source ? La forme *préférée par les développeurs* pour travailler dessus.

Pour en savoir plus : [4] Lisez, étudiez, améliorez, contribuez à des logiciels libres ! [5] ! Soyez professionnels sur les forums (pas de pseudo, votre vrai nom) !

bibliographie

— David SCHMIDT, Brown 1986, *Denotational Semantics: A Methodology for Language Development*
— Michael L. SCOTT, Morgan Kaufmann 2009, *Programming Language Pragmatics*
— Christian QUEINNEC, ParaCampus 2007, *Principes d'Implantation de Scheme et Lisp*

Les livres *Penser autrement l'informatique* (Hermès 1992) et *Artificial Beings* [The Conscience of a Conscious Machine] (Wiley 2009) de Jacques PITRAT sont passionnants et offrent une vision alternative à la conception des langages et des logiciels (dans une vision "strong artificial intelligence"), en insistant sur les aspects introspectifs, réflexifs, méta-connaissances déclaratives.

3. Je suis bienveillant vis à vis de tout language de programmation ayant une implantation en logiciel libre pour Linux. M'en parler préalablement.
4. Bien évidemment, à faire ou à étudier chez soi ou en bibliothèque ou salle machine *après* le cours, pas pendant celui-ci !
5. Vous serez d'abord employé à contribuer à des logiciels -libres ou propriétaires- *existants* (car on ne confie pas le développement ex nihilo d'un nouveau logiciel à un débutant !)

1 à propos des langages de programmation et de leur sémantique

Pour en savoir plus : Pour ceux qui n'ont pas suivi le module *Analyse des Programmes et Sémantiques* en 2013, lire les transparents d'APS, au moins les cours 1, 2, 3, 4, 5, 6

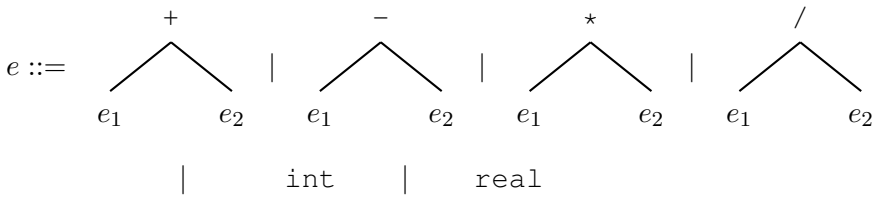
1.1 syntaxes concrète et abstraite

Rappel : la **syntaxe concrète** définit quelles sont les chaînes de caractères (ou représentations textuelles) syntaxiquement acceptables d'un language de programmation : par exemple `2+3*5`. Revoir [6] les analyseurs lexicaux (lexèmes, "lexer program", générateur `flex`, expressions régulières...), classes de grammaires et les analyses syntaxiques (descendantes et ascendantes), LL et LR (générateurs `yacc`, `bison`, `antlr`...), parser. Revoir les automates (à états finis, à pile) et la machine de Turing (et l'indécidabilité du problème de l'arrêt - halting problem, undecidable problem).

La *syntaxe abstraite* traite des **arbres de syntaxe abstrait** (Abstract Syntax Tree). Syntaxe abstraite d'un langage (arithmétique) [7].

$$e \ ::= \ e + e \mid e - e \mid e * e \mid e \,/\, e \mid \texttt{int} \mid \texttt{real}$$

Syntaxe abstraite des arbres :



Représentation par type somme en Ocaml

```
type ast = Plus of ast * ast | Minus of ast * ast
         | Mult of ast * ast | Div of ast * ast
         | Int of int | Real of float ;;
```

et par une union discriminée en C11 :

```
enum kind_en { Nothing, Plus, Minus, Mult, Div, Integer, Real };
struct ast_st {
  enum kind_en kind;
  union {
    struct ast_st *left, *right; // for Plus, Minus, Mult, Div
    int num; // for Integer
    double flo; // for Real
  };
};
```

6. Et relire, par exemple, les articles pertinents de Wikipedia !

7. Tiré des transparents de cours de Jacques Malenfant `http://pagesperso-systeme.lip6.fr/Jacques.Malenfant/`, que j'ai enseigné en 2013, *Analyse des Programmes et Sémantiques*.

Q : quelle hiérarchie de classe Java pour représenter ces AST ?

Domaine de Scott pour les AST :

$$A = \{+ , - , * , / \} \times A \times A \oplus \mathbb{Z} \oplus \mathbb{R}$$

(attention, ça ne peut pas être des ensembles classiques, par raison de cardinalité)
Note : on peut évidemment remplacer $\{+ , - , * , / \}$ par $\{1, 2, 3, 4\}$

1.2 Pile d'appel, cadre d'appels et continuations

Rappel : les ordinateurs *actuels* ont une **pile d'appel** (call stack), matérialisée au moins par un registre de sommet de pile (stack register ou *top of stack* `%esp` sous Linux/x86-64). Chaque appel de fonction s'exécute dans son **cadre d'appel** (call frame), parfois dit *cadre d'activation* (activation record), souvent délimité par un autre registre machine (`%ebp`). La pile d'appel est nécessaire pour exécuter des fonctions récursives. Une interruption (interrupt) machine (dans le noyau) ou un signal Unix (dans un processus sous Linux) modifie de façon asynchrone la pile. Lire attentivement `sigaction(2)` et surtout `signal(7)` (à propos des "async-signal-safe functions"). Les conventions d'appels sont spécifiées dans l'Application Binary Interface, elles varient selon les systèmes et les processeurs. Par exemple, voir x86 calling conventions et x86-64 ABI.

Pour en savoir plus : Coder une fonction récursive simple (par exemple la factorielle en *C* dans un fichier `fact.c`). La compiler avec divers degrés d'optimisation (par exemple avec la commande `gcc -Wall -O1 -S -fverbose-asm fact.c` puis `-O2` ou `-O0` au lieu du `-O1`...). Regarder l'assembleur obtenu dans `fact.s`. Recommencer avec des variantes dans le code source (par exemple des impressions de trace), et des compilateurs différents. Regarder aussi les nombreuses representations intermédiaires en utilisant `gcc -fdump-tree-all` (qui produit des centaines de fichiers de dump !). Dérouler pas à pas l'exécution de cette fonction dans un débogueur (comme `gdb`). Observer le cadre d'appel, les addresses des variables locales, etc... Essayez aussi de désactiver l'ASLR. Découvrir et experimenter les primitives `__builtin_return_address` et `__builtin_frame_address` de `gcc`.

Une occurrence d'appel (une ligne de code avec un appel dans un programme, par exemple en C, Java, Ocaml...) est un **appel récursif terminal** ("tail-recursive call" ou "tail call") si cet appel est la dernière étape de la fonction appelante. Le caractère récursif terminal d'un appel est une propriété *syntaxique*. Un appel récursif terminal ne requiert pas de cadre d'appel spécifique à la fonction appelée (qui peut écraser le cadre d'appel de la fonction appelante.). Il peut donc être réalisé comme un saut avec arguments ("jump with arguments"), par exemple passé dans des registres. Dans la plupart des langages fonctionnels (Ocaml, Haskell, ...) l'appel récursif terminal est la principale *construction **itérative***. Hélas, le code octet (byte code) de la JVM ne connait pas l'appel récursif terminal (mais c'est possible).

Pour en savoir plus : Étudier le code source de quelques fonctions de votre logiciel libre favori et y trouver tous les appels terminaux.

Remarque : l'élimination des récursions des appels terminaux est requise dans certains langages (Ocaml, Scheme), optionnelle mais fréquente dans d'autres (Common Lisp), plus rares dans d'autres implantations de language (par exemple, et *dans certains cas* seulement, les versions récentes des compilateurs GCC, pour C, C++, Ada, Fortran, Go ...).

La *continuation* [8] est une représentation abstraite de l'état de contrôle d'un programme en cours d'exécution, c.à.d du contexte d'exécution, autrement dit de la pile d'appel.

Pour en savoir plus : Jouer avec le `call/cc` de Scheme. Prenez le temps de l'expérimenter. Voir aussi les continuations partielles (delimited continuations) et les opérateurs `shift` et `reset`.

La transformation à passage de continuations consiste à transformer le source d'un programme en CPS = *Continuation Passing Style* : chaque fonction reçoit en plus explicitement en argument supplémentaire sa continuation (comme une fonction traitant son résultat). Voir aussi CPC (Continuation Passing C).

2 λ-calcul et introduction à la sémantique formelle

2.1 rappels sur le λ-calcul

Nous utiliserons, pour noter les fonctions, une λ-notation : $\lambda x.e$ est la fonction de paramètre x et de corps e ; $\lambda x.\lambda y.e$ est la fonction à deux paramètres x et y, et de corps e.

La λ-notation des fonctions est à la base d'une modélisation très générale de l'écriture et l'évaluation des expressions fonctionnelles : le λ-calcul.

Les expressions, on dit les *termes* du λ-calcul (ou λ-termes) sont définis ainsi : on considère un ensemble infini dénombrable $\mathcal{X}$ de symboles dits de *variables* ; l'ensemble des termes est le plus petit ensemble qui satisfasse

1. les variables sont des termes.
2. si x est une variable et t un terme alors $\lambda x.t$ est un terme.
3. si t et u sont deux termes alors $(t\ u)$ est un terme.

Le terme $\lambda x.t$ est appelé une *abstraction*, le terme $(t\ u)$ est une *application*.

On pourra utiliser les abréviations suivantes :

— $(t\ u_1\ u_2)$ pour $((t\ u_1)\ u_2)$

— $\lambda x_1.x_2.t$ pour $\lambda x_1.\lambda x_2.t$

On pourra également mettre des parenthèses autour d'une abstraction si cela peut faciliter la lecture, sans qu'il faille y voir une application.

Outre ces constructions de base, on peut ajouter à la syntaxe des λ-termes des fonctions ou opérations pédéfinies. Par exemple $\lambda n.n^2$ est la notation de la fonction d'élévation au carré. Naturellement, on ajoutera également au λ-termes les constantes numériques. Ainsi, on pourra écrire l'application $(\lambda x.x^2\ 42)$.

α-équivalence Dans l'écriture $\lambda x.t$, la variable x est *liée* à la manière dont une variable x est liées dans la formule $\forall x.\Phi$. Les deux fonctions $\lambda x.t$ et $\lambda y.t[y/x]$ sont deux fonctions *équivalentes*

— si $t[y/x]$ est le terme obtenu en remplaçant x par y dans t ;

— et si y n'a pas d'autre occurence dans t.

β-réduction La β-réduction est la modélisation du principe d'évaluation des applications de fonctions par la substitution :

— $(\lambda x.t\ u)$ (lire : $\ll\lambda x.t$ appliqué à $u\gg$) se réduit en $t[u/x]$;

— si t se réduit en t' alors $(t\ u)$ se réduit en $(t'\ u)$;

— si u se réduit en u' alors $(t\ u)$ se réduit en $(t\ u')$;

— si t se réduit en t' alors $\lambda x.t$ se réduit en $\lambda x.t'$.

La règle principale est la première. Une application de la forme $(\lambda x.t\ u)$ est appelée un *redex*.

8. livre d'Andrew APPEL : *Compiling with Continuations*, Cambridge University Press, 1992 ; Article d'Andrew KENNEDY : *Compiling with Continuations, Continued*, ICFP2007

On notera $t \rightsquigarrow v$ pour $\ll t$ se réduit en $v \gg$

Comme dans le renommage des variables liées, il faut prendre garde, lors de la substitution d'un terme à une variable, qu'aucune des variables du terme u ne devienne liée dans le résultat de la substitution. Techniquement, on obtient ce résulat en renommant systématiquement les variables liées du terme où l'on remplace x lors du processus de susbtitution :

$$(\lambda y.t)[u/x] = \lambda z.(t[z/y][t/x])$$

en choisissant z n'ayant aucune occurence ni dans t ni dans u.

Pour en savoir plus : lire les wikipages sur λ-calculus, Currying (currification), Closure (clôture ou fermeture). Lire *tutorial introduction to the Lambda calculus* de Raul ROJAS, 1997.

Un **environnement** associe aux variables une propriété ou une valeur. Par exemple, l'environnement de typage (typing environment) associe un type à chaque variable.

2.2 premier exemple de sémantique formelle : expressions arithmétiques

En l'absence de variables, la *sémantique* (semantics) des expressions arithmétiques se réduit à une *sémantique opérationnelle* (operational semantics) qui décrit comment calculer la valeur notée entre doubles crochets [[E]] d'une expression E donnée comme un AST :

$$\begin{array}{rcl}
[[\texttt{num}_x]] & = & x \\
[[e_1 + e_2]] & = & [[e_1]] +_{\boldsymbol{R}} [[e_2]] \\
[[e_1 - e_2]] & = & [[e_1]] -_{\boldsymbol{R}} [[e_2]] \\
[[e_1 * e_2]] & = & [[e_1]] *_{\boldsymbol{R}} [[e_2]] \\
[[e_1 \ / \ e_2]] & = & \bot \ \textbf{si} \ [[e_2]] = 0 \\
[[e_1 \ / \ e_2]] & = & [[e_1]]/_{\boldsymbol{R}}[[e_2]] \ \ \textbf{si} \ [[e_2]] \neq 0
\end{array}$$

On peut comprendre le bottom $\bot$ comme une valeur indéfinie ou un cas d'erreur.

Pour en savoir plus : lire les wikipages sur lattice (treillis), ontology (ontologie), abstract interpretation (interprétation abstraite), undefined behavior (comportement indéfini).

projet informatique 1 : améliorer `sash`

Appréhender (on commence en TP) le stand-alone shell de David BELL, à télécharger en **`http://members.tip.net.au/~dbell/programs/sash-3.7.tar.gz`** (puis `tar xzvf sash-3.7.tar.gz`, etc...)

Le but du projet est d'ajouter à ce shell l'évaluation d'expressions arithmétiques. Ainsi on voudrait que `-echo $[1+2*3]` ou `-arith '1+2*3'` affiche `7`, et qu'après `setenv DEUX 2` la commande `-echo $[1+$DEUX]` affiche `3`, etc... On codera en C ou C++ [9], si possible C++11 (voir `cplusplus.com` et `cppreference.com` etc...) compilé avec GCC 4.8

1. améliorer le code pour qu'il n'y ait aucun avertissement avec `-Wall -Wextra`

9. Ou peut-être un autre language, si vous êtes capable de le lier à `sash` dans le même processus, et en me demandant préalablement...

2. modifier le `Makefile` pour ajouter au moins votre fichier `eval-upmc.cc` en C++11 (ou C, etc...) ; corriger le `CFLAGS` (pour avoir `-g` au lieu du `-O3`) et effacer le `LDFLAGS` (sans `-s` ni `-static`) ; astuces `make -p` ou `remake -x`
3. améliorer la fonction `makeArgs` dans `utils.c` de sorte que `-echo $HOME et $DEUX depuis $$` expanse comme il faut.
4. définir la représentation en C++ de l'arbre de syntaxe abstrait des expressions arithmétiques ; la documenter succintement (en anglais) en quelques lignes dans votre `README-UPMC` ; quels sont les cas d'erreur d'évaluation ? Comment allez vous les traiter ?
5. coder l'évaluateur d'expression (il faudra aussi modifier `makeArgs`)
6. au choix **l'un des points suivants** [10]
   (a) coder, en utilisant la librarie `jsoncpp`, la sérialisation (serialization) et la déserialisation au format JSON de vos expressions arithmetiques. Par exemple ajouter une commande prédefinie (shell builtin) `jsonast` de sorte que `-jsonast 1+2*3` affiche quelque chose de similaire à { `"op":"add", "left": 1, "right": { "op": "mult", "left": 2, "right": 3 }}`
   (b) coder un simplificateur d'expressions pour les règles $\alpha+0 \to \alpha$, $\alpha*1 \to \alpha$, $\alpha-\alpha \to 0$, $\alpha/\alpha \to 1$
   (c) coder une primitive `-test`, surtout pour les tests arithmetiques des shells (comme `-test $DEUX -gt 1` ...). S'inspirer du test usuel des shells POSIX voir `test(1)`.
   (d) coder un mini-interprète [11] de fonctions arithmetiques : après `-defunarith fact(n) if n<=0 then 1 else n*fact(n-1)` on aurait `-echo $[fact(3)]` qui affiche `6`.
   (e) améliorer l'évaluateur arithmétique pour qu'il traite "intelligemment" [12] les chaînes de caractères non numériques, en documentant.
   (f) ajouter d'autres types de données (en utilisant s'il y a lieu les containers de C++11) à `sash` - par exemple des tableaux, des listes, des fermetures (ou valeurs fonctionnelles), etc...
   (g) brancher l'interprète `lua` (voir `lua.org`) dans `sash`
   (h) me proposer d'autres extensions de `sash`, les implémenter et les documenter après mon accord
7. Bonus à ceux qui font (bien) plus d'une extension ci-dessus.

Il faut travailler selon les usages du logiciel libre.

à rendre par mél avant dimanche 24 novembre 2013 prochain 20h00 heure de Paris

10. Indiquez votre choix tout de suite ; je peux vous forcer la main si les choix sont mal répartis
11. Vous pouvez adopter la syntaxe que vous voulez, en la documentant, tant que la factorielle puisse être définie récursivement et que votre syntaxe abstraite étende la précédente
12. À vous de définir et de documenter comment ! Le traitement du + ou $\star$ de JavaScript ou de Java pourrait vous inspirer.