Analyse des Programmes et Sémantique (1)

Prof[r] Jacques Malenfant

(cours de J.Malenfant modifié et enseigné en 2013 par Basile Starynkévitch)

janvier-avril 2013

MI030 - APS

Cours (en M1) du Professeur Jacques Malenfant
http://pagesperso-systeme.lip6.fr/Jacques.Malenfant/ Professeur en informatique au Laboratoire d'Informatique de Paris 6

Enseigné en 2013 par Basile Starynkevitch
http://starynkevitch.net/Basile/
basile@starynkevitch.net & basile.starynkevitch@cea.fr
ingénieur chercheur au CEA, LIST - http://www-list.cea.fr/
travaille sur gcc-melt.org

♠ Nota Bene

Les transparents à fond rose (pages numérotées ♠) et les mots ♠ signalés ainsi ♣ sont de Basile Starynkevitch (dont les opinions n'engagent que lui) ♠

La plupart des transparents sont [recopiés de ceux] de J.Malenfant 2012, que je remercie.

Organisation et plan de cours

- Spécialité STL : préparatoire à la fois aux parcours M2 APr, TA et APi, surtout pour les étudiants versés dans la programmation en général.
- Caractère à la fois théorique (formalismes) et pratique (utilisations concrètes, sémantiques exécutables, démarches intégrées dans des processus d'implantation vérifiés).
- 10 semaines :

  Lundi, 13h30–15h30 : 2h cours magistraux.
  Mercredi, 13h30–15h30 : 2h de travaux dirigés.
  Mercredi, 15h45–17h45 : 2h de travaux sur machines encadrés.

♠ Je (= Basile S.) serai absent la semaine du 3 au 9 mars 2013
(TD & TME assurés) ♣

Évaluation

- Un examen réparti en deux :
  - un examen réparti de mi-semestre, pour 40% de la note finale, et
  - un examen réparti de fin de semestre, pour 60% de la note finale.

Cours 1

Introduction

Languages de programmation I

Définition : language de programmation

Médium permettant de communiquer avec l'ordinateur à propos d'un calcul à réaliser, ou encore entre humains sur des algorithmes de calcul

niveau lexicographique : définit tout ce qui concerne le vocabulaire autorisé par le langage (mots, orthographe, ...) ainsi que sa ponctuation. **Exemple** *(langue naturelle)* : *"Le chat mange la souris."* $\Rightarrow$ Le/chat/mange/la/souris.

analyse lexicographique : fait la reconnaissance des mots à partir de la séquence de caractères qui forme le programme.

Languages de programmation II

niveau syntaxique : définit l'agencement des mots pour obtenir des phrases bien formées ; la syntaxe définit également les relations entre les phrases et leurs constituants, donnant ainsi une description structurelle de chacun des types de phrases.

Exemple : *S=Le/chat, V=mange, C=la/souris*

analyse syntaxique : fait le reconnaissance des phrases du programme, et construit les relations d'imbrication.

Languages de programmation III

niveau sémantique : signification et propriétés du programme, c'est-à-dire ce qu'il calcule de même que toutes les propriétés qu'on peut en déduire (typage, exactitude, vivacité, mais aussi utilisation des ressources, ...).
Exemple : pourquoi *Le/chat/mange/la/souris* et pas *La/souris/mange/le/chat* ?
définition inductive : part de la signification ou de la preuve de la propriété pour les instructions et expressions élémentaires, puis construit la signification ou prouve la propriété pour les phrases composées et ultimement pour les programmes.

Languages de programmation IV

niveau pragmatique : aspects de la signification qui ne peut être comprise qu'en connaissant le contexte de son emploi

exemple en programmation : Le `malloc` du language *C* renvoie un pointeur vers une zone de mémoire "neuve" mais peut échouer (en renvoyant `NULL`). Le programmeur s'attend à ce que dans les conditions usuelles et raisonnables d'utilisation, `malloc` réussisse le plus souvent [1]. Une implémentation de `malloc` qui échoue toujours en renvoyant `NULL` est conforme à la lettre, mais pas à l'esprit, du standard *C99* ; mais en 2013 sur son *ordinateur portable personnel* un appel à `malloc(100LL<<30))` [demandant 100 Go] devrait généralement échouer.... [2]

AMHA, il ne faut pas chercher à formaliser les aspects pragmatiques de manière rigoureuse, mais les conserver à l'esprit.

---

1. Mais il faut *toujours* tester que `malloc` n'échoue pas...
2. Toutefois, chercher `linux memory overcommit` sur le Web... ;

et sur un super-calculateur on peut allouer une zone de 100Go.

Languages de programmation V

♠ Attention

Les distinctions entre niveaux lexical, syntaxique, sémantique, pragmatique ne sont pas si aisées que ça en pratique. ♠

Exemple : influence des déclarations, notamment `typedef` en *C*, sur l'analyse "lexicale"

Voir `http://en.wikipedia.org/wiki/Undefined_behavior` et toutes les occurrences d'*undefined behavior* dans le standard *C2011*

Rêveries :

- Pourquoi un programme source serait un texte,
  et pas un *hypertexte* ou un *dessin* ?
- Pourquoi les outils & théories traitent si peu l'*évolution* des programmes sources ?
- Pourquoi l'ordinateur travaille si peu pour le développeur ?
  (durant le codage, l'ordinateur passe son temps à nous attendre ; il travaille moins pour nous développeurs que pour les comptables !)

Spécification des langages de programmation I

Définition : Spécification d'un language de programmation

Définition précise, structurée, voire formelle du langage à la fois dans la forme des programmes autorisés, dans la signification de ces derniers en termes de calculs réalisés, et des propriétés qui peuvent être observées sur eux.

niveau lexicographique : il

- définit les *lexèmes* du language [3]
- utilise le plus souvent des *expressions regulières*

`analex : Char* → Lexème*`

3. ♠ Traitement des commentaires :
ignorés par le compilateur `javac`, pas le générateur de documentation `javadoc` ou `doxygen` ;
chercher `Frama-C ACSL` sur le web... ♣

Spécification des langages de programmation II

niveau syntaxique : ♠ distinguer ♣

syntaxe concrète : *grammaires indépendantes du contexte*

syntaxe abstraite : *grammaires abstraites, definissant les arbres de abstraite*

analyse syntaxique : `anasyn : Lexème* → AST`

niveau sémantique : pour la signification, on utilise

- les sémantiques discursives,
- les sémantiques opérationnelles classiques,
- les sémantiques opérationnelles structurelles,
- les sémantiques axiomatiques,
- les sémantiques dénotationnelles,
- les sémantiques algébriques,
- ...

Spécification des languages de programmation III

Niveau sémantique : pour les propriétés ♠ sémantiques ♣ on utilise :

- des grammaires attribuées
- des **analyses dirigées par la syntaxe** (dont l'interprétation abstraite)
- des analyses dirigées par le flôt de contrôle [4]
- ...

4. ♠ chercher `model checking program analysis` sur le Web ♣

Syntaxe concrète et syntaxe abstraite I

Définition : Syntaxe concrète

Définition complète et non-ambigüe de l'ensemble des chaînes de caractères formant des programmes syntaxiquement corrects.

Définition : Syntaxe abstraite

Définition des arborescences d'instructions et d'expressions formant l'ensemble des programmes exprimables.

♠ AST = "abstract syntax trees" [5] ♣

5. ♠ pensez toujours AST, même pour des tâches "simples" comme générer des fragments de programmes *C* ou *Java* ♣

Syntaxe concrète et syntaxe abstraite II

- La **syntaxe concrète** se focalise sur la formation et l'agencement des lettres en mots, puis des mots en phrases, de manière à former des programmes syntaxiquement corrects. Elle est spécifiée par une *grammaire concrète*.
- La **syntaxe abstraite** se focalise sur les programmes exprimables dans le langage en termes de constructions imbriquées formant une arborescence. Elle est spécifiée par une *grammaire abstraite*.

♠ Les outils logiciels (compilateurs [6], analyseurs statiques) traitent fondamentalement la syntaxe abstraite.

(les analyses lexicale et syntaxique sont *faciles* [7] ! c'est la *sémantique* qui est difficile !) ♣

6. ♠ Dans le compilateur *GCC* l'analyse syntaxique est une petite part ; l'essentiel de GCC malaxe et pétrit des AST et autres représentations internes... ♣
7. ♠ Il existe beaucoup d'outils générant le code "source" d'analyseurs syntaxiques : `flex` & `bison` pour générer du *C*, *ANTLR* pour *C++* ou *Java*, `menhir` pour du *Ocaml*, etc. Les générateurs sémantiques sont moins usuels (`iburg`) ♣

Grammaire concrète et analyse statique I

Définition : Grammaire concrète

Ensemble de règles de construction de phrases d'un langage qui permettent de construire une dérivation pour chacune des phrases acceptées par la grammaire.

Définition : Analyse syntaxique

Procédé par lequel une séquence de lexèmes est reconnue comme faisant partie du langage défini par une grammaire concrète, en produisant une dérivation, et ayant éventuellement pour résultat un arbre de syntaxe abstraite du programme.

Grammaire concrète et analyse statique II

- Si le modèle formel de calcul intrinsèque à l'analyse lexicale est celui des *automates d'états finis*, celui de l'analyse syntaxique est l'*automate à pile*.
- Ce qui différentie les automates à pile des automates d'états finis est leur capacité à reconnaître les structures récursivement imbriquées (alors que les AEF ne peuvent reconnaître que les structures répétitives).
- Plus précisément, l'analyse syntaxique des langages dits indépendants du contexte est fondée sur les grammaires dîtes indépendantes du contexte.
- L'indépendance par rapport au contexte signifie simplement que la forme d'une phrase est toujours la même, peu importe le contexte où elle apparaît dans la grammaire, et donc dans les phrases acceptées par cette grammaire.

♠ Beaucoup de languages de programmation ont des aspects contextuels.
Quand ceux-ci sont prédominants (e.g. *C++2011*) l'analyse syntaxique est complexe et coûteuse à réaliser, sans pour autant faciliter la lisibilité humaine du code ! ♣

Plus formellement ....

Défintion formelle : **grammaire non-contextuelle**

C'est ("context-free grammar") un quadruplet $G = (V, T, P, S)$ où :

- V : est un ensemble fini de *variables* ou *non-terminaux*
- T : est un ensemble fini de *terminaux*, c'est-à-dire les mots du langage,
- P : est un ensemble fini de *règles de production* de la forme $A \rightarrow \alpha$ où A est un symbole non-terminal et α est une séquence ♠ finie ♣ de symboles terminaux et non-terminaux,
- S : est un symbole non-terminal appelé *axiome* à partir duquel toutes les chaînes du langage sont engendrées par application des règles de production.

♠ Chercher sur le Web `Chomsky innateness hypothesis` et `Whorf-Sapir hypothesis`, vidéo `Chomsky: Grammar, Mind and Body` ♣

Exemple : expressions arithmétiques simples

♠ (slide légèrement modifié par Basile S.) ♣

V : les *3 variables* ou *non-terminaux* sont E [expressions], T [termes], F [facteurs] (elles apparaissent à **gauche** des règles).

T : les *7 terminaux* sont $\{\texttt{(}, \texttt{)}, \texttt{int}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}$; ils n'apparaissent pas à gauche des règles

P : les *8 règles de production*

$$E := E\texttt{+}T \mid E\texttt{-}T \mid T$$
$$T := T\texttt{*}F \mid T\texttt{/}F \mid F$$
$$F := \texttt{(}\,E\,\texttt{)} \mid \texttt{int}$$

S : l'axiome est E (premier non-terminal défini, "start symbol")

♠ Devinette : Quelle est la/une grammaire non-contextuelle des grammaires non-contextuelles ?
Chercher `Backus-Naur Form` sur le web. ♣

Dérivation I

Définition : Dérivation

Une dérivation (notée $\Rightarrow$) est constituée d'une série d'étapes où on applique une règle de production pour remplacer un non-terminal par l'une des parties droites de ses règles de production.

- Une *phrase* est une séquence de symboles terminaux (ou mots).
- Une dérivation partant de l'axiome et arrivant à une phrase montre que cette phrase appartient au langage défini par la grammaire.
- Au cours d'une dérivation, on manipule des *protophrases*, c'est-à-dire des phrases en devenir. Une protophrase est une séquence de symboles terminaux et non-terminaux.

Dérivation II

Exemple

$$
\begin{aligned}
E &\Rightarrow T \Rightarrow T*F \Rightarrow F*F \Rightarrow (E)*F \Rightarrow (E+T)*F \\
&\Rightarrow (T+T)*F \Rightarrow (F+T)*F \Rightarrow (5+T)*F \\
&\Rightarrow (5+F)*F \Rightarrow (5+2)*F \Rightarrow (5+2)*3
\end{aligned}
$$

♠ Donc `(5+2)*3` est une expression arithmétique. Noter que `5` $\in$ `lit` etc... (abusivement) ♣

♠ Prenez le temps chez vous de coder (récursivement) un évaluateur d'expressions arithmétiques [=calculette] ♣

Langage accepté par une grammaire

- Le langage accepté par une grammaire est l'ensemble des phrases pour lesquelles il est possible de construire une dérivation depuis l'axiome.
- Le problème de la reconnaissance des phrases est donc celui de la construction d'une dérivation.
- On voit cependant qu'à chaque étape de dérivation, on doit faire deux choses :
  - choisir le non-terminal à remplacer,
  - choisir la règle de production à utiliser pour faire ce remplacement.
- Toute la difficulté de l'analyse syntaxique se concentre dans ces deux questions, auxquelles chaque algorithme d'analyse apporte ses réponses.
- Le choix du non-terminal à remplacer aux différentes étapes de dérivation peut être facilement réglé en appliquant une décision systématique.

♠ On a en pratique intérêt à limiter la grammaire du language, ou à coder des heuristiques, pour en faciliter l'effcacité de l'analyse ; on veut éviter une complexité exponentielle ! ♣

Choix du non-terminal à remplacer

Définition : Dérivation à gauche

Une dérivation à gauche (notée $\underset{g}{\Rightarrow}$) est une dérivation dans laquelle, à chaque étape de dérivation, on choisit toujours le non-terminal le plus à gauche dans la protophrase courante.

Définition : Dérivation à droite

Une dérivation à droite (notée $\underset{d}{\Rightarrow}$) est une dérivation dans laquelle, à chaque étape de dérivation, on choisit toujours le non-terminal le plus à droite dans la protophrase courante.

Exemple ♠ de dérivation à droite ♣

Considérons la grammaire :

$$\begin{array}{lcl} S & \rightarrow & \mathtt{a}A B\mathtt{e} \\ A & \rightarrow & A\mathtt{bc} \mid \mathtt{b} \\ B & \rightarrow & \mathtt{d} \end{array}$$

La dérivation à droite de la phrase `abbcde` est :

$$S \underset{d}{\Rightarrow} \mathtt{a}A\underline{B}\mathtt{e} \underset{d}{\Rightarrow} \mathtt{a}\underline{A}\mathtt{de} \underset{d}{\Rightarrow} \mathtt{a}\underline{A}\mathtt{bcde} \underset{d}{\Rightarrow} \mathtt{abbcde}$$

Choix de la règle

- Le choix de la règle à utiliser pour remplacer le non-terminal est beaucoup plus épineux car un mauvais choix peut mener à une impasse dans la dérivation alors qu'il peut exister un bon choix permettant de dériver la phrase.
- Dans le cas général, on peut utiliser un algorithme avec *retour arrière*[8] qui revient sur les choix aux différentes étapes précédentes si on ne réussit pas à dériver la phrase soumise à l'analyse.
- Comme cette solution est généralement trop coûteuse, on cherche plutôt à *déterminer la bonne règle* en regardant les symboles apparaissant dans le phrase pour guider le choix.
- Lorsqu'on part de l'axiome pour aller vers la phrase, on peut par exemple comparer le prochain symbole à générer dans la phrase et voir quelle règle permet de dériver ce symbole.

8. ♠ D'où la complexité exponentielle en la longueur de la phrase à analyser ! ♣

Principales approches d'analyse syntaxique

Analyse descendante : cette forme d'analyses part de l'axiome et essaie de dériver la phrase à analyser.

Principaux algorithmes :

- descente récursive, avec ou sans retour arrière [9] ;
- analyses LL(k), $k \geq 1$.

Analyse ascendante : cette forme d'analyses part de la phrase à analyser et essaie de revenir à l'axiome [10].

Principaux algorithmes :

- algorithme d'Earley ;
- analyses SLR, LR(k) et LALR(k), $k \geq 0$.

9. ♠ Générateur *ANTLR*, ... ♣
10. ♠ Générateurs `yacc`, `bison`, `menhir`, etc... ♣

Grammaire abstraite et arbre de syntaxe abstraite I

Définition : Arbre de syntaxe abstraite

Arbre représentant le contenu d'un programme sous la forme d'instructions et d'expressions où les relations parents-fils de l'arborescence représentent les relations d'imbrications entre instructions et sous-instructions, et expressions et sous-expressions.

Définition : Grammaire abstraite

Grammaire engendrant les arbres de syntaxe abstraite admissibles pour un langage donné, par la définition des types de noeuds (non-terminaux et terminaux) et des sous-arbres possibles pour chacun de ceux-ci.

Grammaire abstraite et arbre de syntaxe abstraite II

- La syntaxe abstraite ne s'intéresse plus à la représentation du programme dans un fichier ♠ ou une chaîne de caractères ♣, mais bien à son contenu et à sa structure.
- Elle s'intéresse donc au concept ♠ ou aux représentations **internes** ♣ derrière les instructions plutôt qu'à leur forme externe : le concept d'instruction d'affectation, de répétition, d'alternative, ...
- Une grammaire abstraite définit aussi une structure de données de type arborescence, et l'ensemble des arbres (programmes) pouvant être créés grâce à cette structure de données.

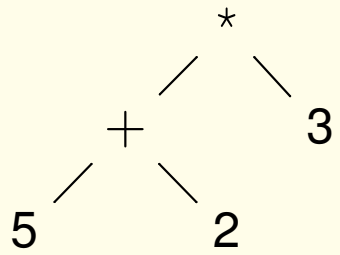
Exemple

- La grammaire abstraite des expressions arithmétiques :

$$e \quad ::= \quad e + e \mid e - e \mid e * e \mid e / e \mid int$$

♠ en toute rigueur, les signes $+$, $-$ ... ne sont pas des lexèmes, mais des discriminants d'arbre (ou des "classes") ♣

- L'arbre de syntaxe abstraite pour l'expression `(5+2)*3` :



Réalisation informatique d'un arbre de syntaxe abstraite I

En Java ou C++, une super-classe abstraite commune `class Arbre` avec des sous-classes `class NoeudSomme`, `class NoeudDifference`, `class NoeudProduit`, `class NoeudQuotient`, qui contiennent [11] des [pointeurs vers] des sous-`Arbre`s, et une `class FeuilleNombre` contenant un nombre.

En Ocaml un type somme comme

```
type arbre_t =
    FeNombre of int
  | NdSomme of arbre_t *  arbre_t
  | NdDifference of arbre_t *  arbre_t
  | NdProduit of arbre_t *  arbre_t
  | NdQuotient of arbre_t *  arbre_t
```

11. Probablement par une classe abstraite intermédiaire `class NoeudBinaire`

Réalisation informatique d'un arbre de syntaxe abstraite II

En C2011, une structure avec une union discriminée à la main

```
enum genre_en { FNombre, NSomme, NDifference, NProduit, NQuotient };
struct arbre_st {
  enum genre_en ar_genre;
  union {
    // quand ar_genre == FNombre  :
     int ar_num;
    // pour  NSomme, NDifference, NProduit, NQuotient :
     struct {
       struct arbre_st *ar_gauche;
       struct arbre_st *ar_droite;
     };
  };
};
```

Réalisation informatique d'un arbre de syntaxe abstraite III

En réalité, le diable est dans les détails :

- gestion de la mémoire
- representation optimisée (car il y a des centaines de milliers de noeuds d'arbres)
- conserver l'origine d'un arbre
  au moins, position : nom de fichier source, ligne, colonne
- robustesse aux erreurs (y compris dans le programme source)
  diagnostics pour l'utilisateur

Suggestion : coder chez vous un évaluateur d'AST pour expressions

Sémantique discursive

Définition : Sémantique discursive

Spécification informelle par laquelle la signification des contructions du langage est donnée sous la forme d'explications en langue naturelle [a]

a. ♠ Souvent en anglais, avec un sens précis pour les verbes *should*, *can*, ... ♣

- Étape première et souvent incontournable pour donner une première définition d'un langage.
- Bon médium de communication avec les utilisateurs.
- Toute l'ambiguïté de la langue naturelle dans la communication avec les implanteurs.

♠ C'est pourquoi les standards des languages occasionnent des débats homériques. ♣

Sémantique opérationnelle classique

Définition : Sémantique opérationnelle classique

Spécification systématique consistant à fournir une implantation précise sur une plate-forme donnée, pouvant servir de référence.

- Typiquement un interprète écrit dans un langage de haut niveau bien connu, mais peut aussi être par traduction (compilation) vers un langage bien connu.
- Force à passer par tous les aspects du langage.
- Pas d'ambiguïté, mais il y aura nécessairement des erreurs dans l'implantation.
- Nécessite l'accès à la plate-forme pour être utilisable.
- La sémantique du langage qu'on cherche à définir ne pourra pas être moins ambigüe que la sémantique du langage utilisé pour écrire l'interprète ou vers lequel on traduit.

Interprètes méta-circulaires

Une forme particulière ♠ de sémantique opérationnelle classique ♣ : les interprètes méta-circulaires

- Un interprète du langage écrit dans le langage lui-même.
- Fournit un bon test de capacité d'expression du langage en montrant qu'il peut servir à sa propre implantation.
- Vu comme un exercice de style intellectuellement et esthétiquement intéressant.
- S'apparente à des techniques d'implantation des compilateurs par auto-amorçage ♠ "bootstrap" ♣
- ♠ N'enpêche pas nécessairement l'ambigüité : "call-by-name vs call-by-value" en Lisp ♣

Sémantique opérationnelle structurelle (SOS) I

Définition : Sémantique opérationnelle structurelle

Utilisant des machines abstraites formalisées, la SOS se fonde sur la syntaxe abstraite du langage pour spécifier formellement la sémantique de chaque construction par des règles empruntées au calcul des séquents décrivant comment la construction change l'état de la machine abstraite.

- Les machines abstraites formalisées se définissent par un état (tuples, listes, ..., bien définis) et un ensemble de relations opérant sur ces structures de données.
- Relativement facile à comprendre, les SOS fournissent un bon médium de communication avec les utilisateurs et les implanteurs.
- Les règles du calcul des séquents s'implantent relativement facilement dans un langage comme Prolog, ce qui fournit un moyen aisé d'obtenir une sémantique exécutable, et de « tester » la sémantique sur des programmes réels. Cela n'apporte aucune garantie formelle d'exactitude de la sémantique, mais rassure très pragmatiquement.

Sémantique opérationnelle structurelle (SOS) II

- Leur définition, fondée sur la grammaire abstraite du langage, se présente donc sous une forme **récursive**, la sémantique d'une construction étant construite à partir de la sémantique de ses sous-constructions.
- Le grand intérêt de cette approche est d'autoriser les preuves de propriétés par **induction structurelle** sur les constructions du langages de manière similaire à l'induction mathématique :
  1. preuve de la propriété sur les constructions de base ; puis,
  2. preuve de la propriété pour chaque construction composée en supposant qu'elle est vraie pour ses sous-constructions.

  C'est une des formes de preuve de programme parmi les plus simples.

Sémantique axiomatique I

Définition : Sémantique axiomatique

Plus abstraite que la sémantique opérationnelle structurelle, la sémantique axiomatique se réfère également à l'état du programme, mais décrit par une *assertion initiale* (*précondition*, représentant les relations devant être vraies au début de l'exécution du programme) et une *assertion finale* (*postcondition*, représentant celles qui doivent être vraies à la fin de l'exécution **si** la précondition était respectée au début).

- L'essence de la sémantique axiomatique consiste à construire l'assertion finale à partir de l'assertion initiale en capturant comment le programme agit pour transformer progressivement la première en la seconde en passant par une série d'assertions intermédiaires.
- Les fondements mathématiques de la sémantique axiomatique se retrouvent donc dans la logique des prédicats.

♠ Chercher `logique de Hoare` sur le Web ♣

Sémantique axiomatique II

- La construction progressive de l'assertion finale est fondée sur la spécification pour chacun des types d'instructions du langage par des pré- et post-conditions représentant une *spécification* pour ces instructions :

$$\{\mathit{Pré\text{-}condition}\}\ \mathit{instr}\ \{\mathit{Post\text{-}condition}\}$$

- Exemple : sémantique de l'affectation

$$\{P[v \to e]\}\ v := e\ \{P\}$$

et sur un cas particulier, on pourrait avoir :

$$\{(\mathtt{k} = 6)[\mathtt{k} \to \mathtt{k}+1]\}\ \mathtt{k} := \mathtt{k}+1\ \{\mathtt{k} = 6\}$$

$$\{\mathtt{k}+1 = 6\}\ \mathtt{k} := \mathtt{k}+1\ \{\mathtt{k} = 6\}$$

$$\{\mathtt{k} = 5\}\ \mathtt{k} := \mathtt{k}+1\ \{\mathtt{k} = 6\}$$

Sémantique dénotationnelle I

Définition : Sémantique dénotationnelle

Approche formelle définissant la sémantique des programmes et des constructions syntaxiques en leur faisant correspondre des objets mathématiques bien définis comme les entiers, les réels, les valeurs de vérité et les fonctions.

- Fondée sur le principe selon lequel les programmes manipulent des représentations syntaxiques d'objets mathématiques.
- L'utilisation de constructions récursives dans les programmes implique que les fondements mathématiques de la sémantique dénotationnelle se trouvent dans la *théorie des fonctions récursives*.
- Peuvent s'implanter dans un langage fonctionnel comme Scheme, ce dernier réalisant opérationnellement une certaine implantation du λ-calcul.

♠ Toujours une idéalisation simplificatrice de l'ordinateur ♣

Sémantique dénotationnelle II

- Le but ultime étant d'associer à tout programme la fonction mathématique qu'il calcule, l'idée est de construire progressivement cette fonction en associant à chaque construction syntaxique du langage une règle donnant l'objet mathématique correspondant à chacune des instances de cette construction pouvant apparaître dans les programmes.
- Un autre principe fondamental de la sémantique dénotationnelle est la définition de règles compositionnelles, c'est-à-dire qui donnent la sémantique d'une construction en fonction de la sémantique de ses sous-constructions et des éléments apparaissant dans la construction, à l'exclusion de toute autre chose.
- Les sémantiques dénotationnelles faisant la correspondance entre de la syntaxe et des objets mathématiques, la présentation des règles de construction met l'emphase sur ce point en encadrant les formes syntaxiques par des doubles crochets :

$$\mathcal{E}[\![e_1 + e_2]\!] = \mathcal{E}[\![e_1]\!] + \mathcal{E}[\![e_2]\!]$$

attention, ce n'est pas le même '+' !

♠ à gauche, le $+$ des arbres de syntaxe abstraite ; à droite, le $+$ des nombres. ♣

Sémantique algébrique

Définition : Sémantique algébrique

Approche formelle définissant la sémantiques des programmes, des constructions syntaxiques, des données et des opérations sur ces données, en leur faisant correspondre une spécification algébrique.

- L'emphase de la sémantique algébrique porte sur la dénomination des sortes d'entités manipulées dans les programmes et des opérations sur ces dernières de manière à utiliser des axiomes algébriques pour décrire leurs propriétés caractéristiques.
- L'idée fondamentale est donc de projeter la syntaxe sur des objets mathématiques riches, dont les propriétés (commutativité, associativité, distributivité, élément neutre, élément absorbant, ...) vont pouvoir être exploitées dans les preuves de programme et les analyses.

Les grammaires attribuées

Définition : Grammaire attribuée

Grammaire où des attributs sont associés aux symboles et où des règles de calcul de ces attributs sont associées aux productions.

- Mécanisme de manipulation et transformation d'arbres utilisé en compilation pour calculer ou vérifier des propriétés sur les arbres de syntaxe abstraite.
  Exemples : vérification des types, génération de code intermédiaire.

♠ attributs hérités $\neq$ attributs synthétisés ♣

Exemple I : Évaluation d'une expression arithmétique

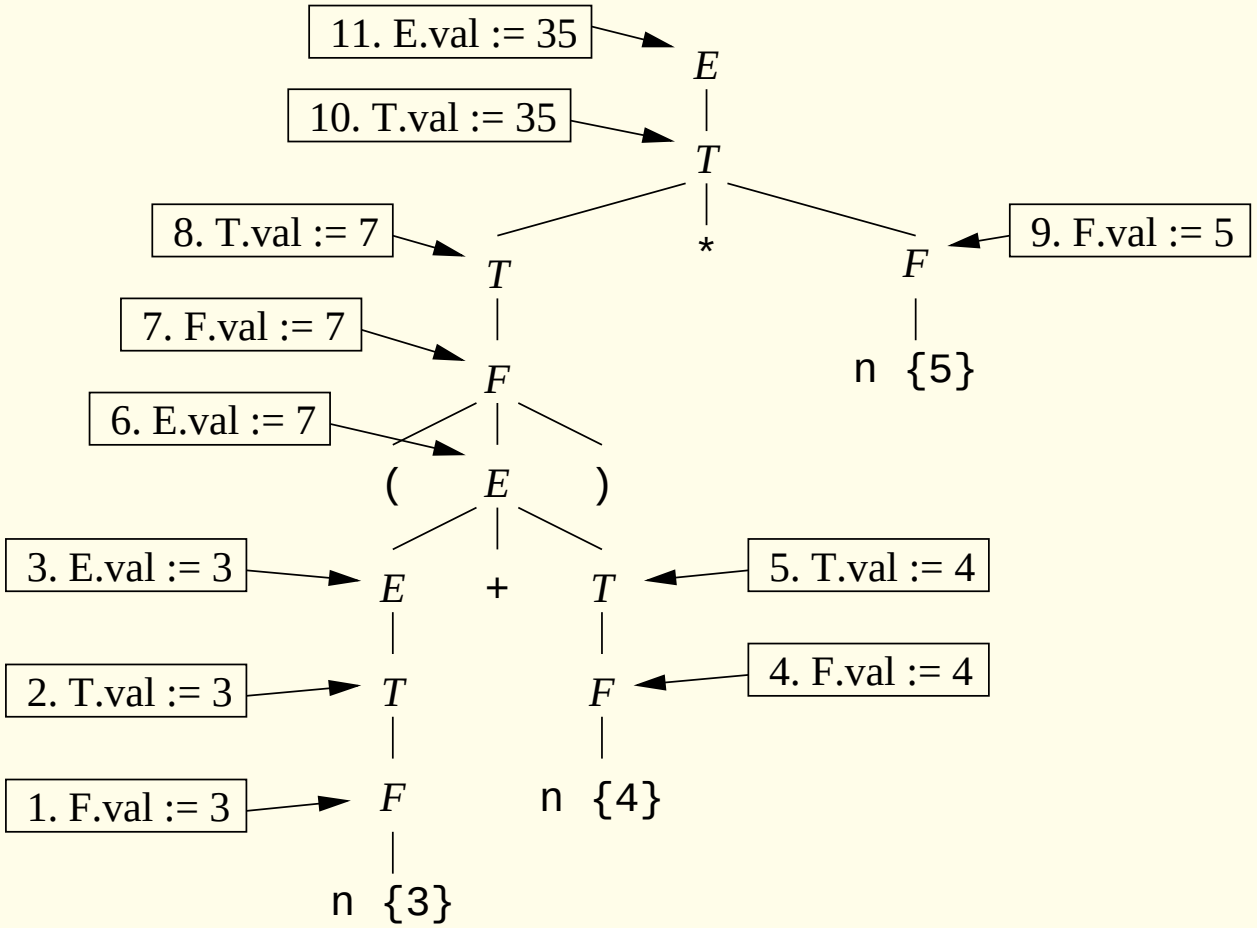
- Objectif : à partir d'une représentation d'une expression arithmétique sous forme d'arbre conforme à une grammaire donnée des expressions, calculer le résultat de l'expression.
- Attributs utilisés :
  - `val` : valeur calculée par la sous-expression engendrée par le non-terminal attribué.
  - `vallex` : valeur lexicale lue pour le symbole terminal `n`
    (cette valeur est obtenue directement lors de la lecture du texte par l'analyseur lexical).
- Règles de calcul des attributs données dans un tableau.

Exemple II : Règles de calcul

$E \rightarrow E_1 + T$	$E.\texttt{val} := E_1.\texttt{val} + T.\texttt{val}$
$E \rightarrow E_1 - T$	$E.\texttt{val} := E_1.\texttt{val} - T.\texttt{val}$
$E \rightarrow T$	$E.\texttt{val} := T.\texttt{val}$
$T \rightarrow T_1 * F$	$T.\texttt{val} := T_1.\texttt{val} \times F.\texttt{val}$
$T \rightarrow T_1 / F$	$T.\texttt{val} := T_1.\texttt{val} \div F.\texttt{val}$
$T \rightarrow F$	$T.\texttt{val} := F.\texttt{val}$
$F \rightarrow (E)$	$F.\texttt{val} := E.\texttt{val}$
$F \rightarrow \texttt{n}$	$F.\texttt{val} := \texttt{n.vallex}$

♠ `.val` est un attribut synthétisé ♣

Exemple III : évaluation des attributs



Analyses dirigées par la syntaxe

- Tout comme la définition de la sémantique des langages, une approche très largement utilisée consiste à définir les analyses par induction sur la structure des instructions et expressions du langage, et donc sur les arbres de syntaxe abstraite.
- Les analyses les plus simples peuvent se réaliser en un seul passage sur l'arbre de syntaxe abstraite du programme ; mais certaines nécessitent plusieurs passages, voire dans certains cas le calcul d'un point fixe (itérer jusqu'à convergence sur un résultat fixe).
- Une forme maintenant bien connue : l'**interprétation abstraite**
  ♠ inventée en 1977 par un couple français : Patrick et Radhia Cousot ♣
- Les formalismes utilisés sont souvent les mêmes que pour la sémantique, c'est-à-dire la logique des prédicats, les fonctions récursives, etc.

Interprétation abstraite

Définition : Interprétation abstraite

Procédé d'analyse sémantique des programmes dirigée par la syntaxe mimant une interprétation (standard) en remplaçant les domaines (concrets) des valeurs par des domaines abstraits et les opérations par des opérations abstraites, définies de telle manière à produire le résultat attendu de l'analyse.

- Domaines sont abstraits de telle manière que :
  - les domaines de valeurs et des structures de données manipulées dans le programme permettent de contenir les informations nécessaires à l'analyse ;
  - le type du résultat représente le résultat de l'analyse.
- Les opérations sont abstraites de manière à calculer les résultats intermédiaires de l'analyse.
- L'analyse est formulée par des règles de calcul pour chacune des instructions et expressions du langage, de la même manière qu'on définit la sémantique des programmes.
- Formulée comme une sémantique dénotationelle.

Exemple d'interprétation abstraite : règle des signes I

- Opération standard : $\times : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$
- Abstraction du domaine : $\mathbb{R} \Rightarrow S$

$$S = \begin{cases} + & si\ r > 0 \\ 0 & si\ r = 0 \\ - & si\ r < 0 \end{cases}$$

♠ On peut avoir envie d'introduire $\top$ ("top") si on ne sait rien du signe de r ♣

Exemple d'interprétation abstraite : règle des signes II

- Abstraction de l'opération : $\times \Rightarrow \times_{abs}$

$\times_{abs}$	+	0	−
+	+	0	−
0	0	0	0
−	−	0	+

  - ♠ Et $\top$ serait absorbant : $\forall \sigma \in S : \top \times_{abs} \sigma = \sigma \times_{abs} \top = \top$ ♣
- Analyse du signe du résultat : *signe* : $S \times_{abs} S \to S$

Opinions sur l'interprétation abstraite

NB : Je (=Basile) connais encore mal l'interprétation abstraite

- papiers français souvent [trop] mathématiques ;
  bons tutoriaux américains
  `http://en.wikipedia.org/wiki/Abstract_interpretation`
- en pratique, on veut que l'interprétation abstraite (i.e. l'analyse statique) termine (même sur un programme "incorrect" qui ne termine pas)
  $\rightarrow$ importance du **"widening"**
- l'interprétation abstraite est comme la prose de M^{r} Jourdain :
  tout le monde en fait sans le savoir
- notions à approfondir : concrétisation, bi-simulation, treillis, correspondance de Galois
- compromis : précision $\neq$ complexité d'analyse
  plein de règlages internes dans les analyseurs statiques
- pleins de domaines (treillis) : intervalles numériques, symboliques, octogones, "shape analysis", ...

Les analyses dirigées par la flôt de contrôle

Définition : Analyses dirigées par la flôt de contrôle

Ensemble d'analyses fondées sur le parcours du graphe de flot de contrôle du programme, obtenu généralement au cours de l'exécution de la partie terminale d'un compilateur, c'est-à-dire lors de la phase initiale de génération de code à partir de l'arbre de syntaxe abstraite.

- Le graphe de flot de contrôle se présente sous la forme de blocs d'instructions séquentielles reliés entre eux par des arcs indiquant l'ordre dans lequel ces blocs peuvent être exécutés.
- Les algorithmes d'analyse propagent généralement les informations à travers ce graphe. Il peut être nécessaire de le parcourir plusieurs fois, voire de calculer un point fixe.
- Historiquement, cette forme d'analyses est apparue avant les analyses dirigées par la syntaxe, pour les besoins de la compilation (allocation de ressources, comme les registres des processeurs, et les optimisations).

Présentation du langage BOPL

- *Basic Object Programming Language*.
- Dû à Palsberg et Schwartzbach :

  Palsberg, J. et Scwartzbach, M. I., *Object-Oriented Type Systems*. Wiley, 1994.
- Comporte les principaux traits objets : classes, variables d'instance, méthodes, typage par les classes, substituabilité, super, self, ...

Brève description du langage BOPL I

- Un programme est constitué d'une séquence de déclarations de classes suivie par un corps, c'est-à-dire une déclaration optionnelle de variables locales suivie d'une séquence d'instructions. L'exécution d'un programme consiste en l'exécution de cette séquence d'instructions, après la création des classes et dans le contexte des variables locales déclarées.
- Une classe comprend un identifiant unique dans le programme, une liste de variables d'instance et une liste de méthodes. Elle peut être définie comme héritant d'une autre classe, par la clause *extends* ; par défaut les classes héritent de la classe `Object`, racine de l'arbre d'héritage.
- Outre `Object`, trois autres classes prédéfinies représentent les types de base du langage : `Bool`, `Int` et `Void`.
- Les variables locales et d'instance, comme les paramètres et la valeur de retour des méthodes, sont typées par les classes du programme (ou les classes prédéfinies).

Brève description du langage BOPL II

- Une méthode possède donc un nom, une liste de paramètres formels typés, un type de retour, et un corps composé d'une déclaration de variables locales suivie d'une séquence d'instructions.
- La redéfinition des variables dans les sous-classes n'est pas admise, mais la redéfinition des méthodes l'est. Lors de la redéfinition, il est possible de modifier les types, la règle applicable étant la contravariance (les types des paramètres formels de la méthode redéfinissante sont des superclasses de celles de la méthode redéfinie, alors que le type du résultat de la méthode redéfinissante doit être une sous-classe du type du résultat de la méthode redéfinie.
- Les instructions comportent l'affectation d'une variable locale, l'affectation d'une variable d'instance, la séquence, l'alternative (if-then-else), la répétition (while), l'instruction de retour du résultat (return) et l'instruction d'écriture sur la sortie standard (writeln).
- Les expressions comportent l'accès aux variables d'instance, l'appel de méthode, le test d'appartenance (*instanceof*), les expressions arithmétiques et les expressions booléennes.
- Les affectations et l'appel de méthode autorisent le principe de substituabilité, c'est-à-dire qu'une instance d'une classe peut toujours être substituée par une instance d'une sous-classe.

Grammaire concrète du langage BOPL I

Program	: :=	**`program`** Classlist Locals Seq
Locals	: :=	ε | **`let`** Vars **`in`**
Classlist	: :=	ε | Classes
Classes	: :=	Class | Classes Class
Class	: :=	**`class`** Id Extends **`is`** Varlist Methodlist **`end`** |
Extends	: :=	ε | **`extends`** Classexp
Classexp	: :=	**`Int`** | **`Bool`** | **`Object`** | **`Void`** | Id
Varlist	: :=	ε | **`vars`** Vars
Vars	: :=	Var | Vars Var
Var	: :=	Classexp Ids **`;`**
Ids	: :=	Id | Ids **`,`** Id

Grammaire concrète du langage BOPL II

Methodlist	: :=	ε | **methods** Methods
Methods	: :=	Method | Methods Method
Method	: :=	Classexp Id **(** Formallist **)** Locals Seq
Formallist	: :=	ε | Formals
Formals	: :=	Formal | Formals **,** Formal
Formal	: :=	Classexp Id
Seq	: :=	**begin** Insts **end**
Insts	: :=	Inst | Insts **;** Inst
Inst	: :=	Id **:=** Exp | Exp **.** Id **:=** Exp | return Exp | **if** Exp **then** Seq **else** Seq | **while** Exp **do** Seq | **writeln (** Exp **)**

Grammaire concrète du langage BOPL III

Exp	: :=	Exp `.` Id | Exp `.` Id `(` Actuallist `)` |
		Exp **`instanceof`** Classexp |
		Exp `+` Term | Exp `-` Term |
		Exp **`or`** Term | Term
Term		Term `*` Fact | Term **`and`** Fact | Fact
Fact	: :=	Fact `=` Basic | Fact `<` Basic | Basic
Basic		**`not`** Exp | Int | Id | **`true`** | **`false`** | **`nil`** |
		`self` | **`super`** | **`new`** Classexp | `(` Exp `)`
Actuallist	: :=	ε | Actuals
Actuals	: :=	Exp | Actuals `,` Exp

Un exemple de programme BOPL

```
program
  class Point is
  vars
     Int x, y ;
  methods
    Point add(Point p)
    let
      Point ret ;
    in
    begin
      ret := new Point ;
      ret.x := (self.x) + (p.x) ;
      ret.y := (self.y) + (p.y) ;
      return ret
    end
  end
  let
    Point p1, p2, p3 ;
  in
  begin
    p1 := new Point ;
    p1.x := 1 ;
    p1.y := 2 ;
    p2 := new Point ;
    p2.x := 5 ;
    p2.y := 10 ;
    p3 := p1.add(p2) ;
    writeln(p3)
  end
```

Grammaire abstraite du langage BOPL I

$$
\begin{array}{rcl}
\textit{program} & ::= & \texttt{program}\ \textit{class}^*\ \textit{var}^*\ \textit{inst} \\
\textit{class} & ::= & \texttt{class}\ \textit{id}\ \textit{cexp}\ \textit{var}^*\ \textit{method}^* \\
\textit{cexp} & ::= & \texttt{cexp}\ \textit{id} \\
\textit{var} & ::= & \texttt{var}\ \textit{cexp}\ \textit{id} \\
\textit{method} & ::= & \texttt{method}\ \textit{id}\ \textit{var}^*\ \textit{cexp}\ \textit{var}^*\ \textit{inst}
\end{array}
$$

Grammaire abstraite du langage BOPL II

$$
\begin{array}{lcl}
\textit{inst} & ::= & \texttt{seq}\ \textit{inst}\ \textit{inst} \mid \texttt{assign}\ \textit{id}\ \textit{exp} \mid \texttt{writefield}\ \textit{exp}\ \textit{id}\ \textit{exp} \mid \\
 & & \texttt{if}\ \textit{exp}\ \textit{inst}\ \textit{inst} \mid \texttt{while}\ \textit{exp}\ \textit{inst} \mid \texttt{return}\ \textit{exp} \mid \\
 & & \texttt{writeln}\ \textit{exp} \\
\textit{exp} & ::= & \textit{int} \mid \texttt{true} \mid \texttt{false} \mid \texttt{not}\ \textit{exp} \mid \\
 & & \texttt{nil} \mid \texttt{self} \mid \texttt{super} \mid \texttt{new}\ \textit{cexp} \mid \texttt{instanceof}\ \textit{exp}\ \textit{exp} \mid \\
 & & \textit{id} \mid \texttt{methodcall}\ \textit{exp}\ \textit{id}\ \textit{exp}^{*} \mid \texttt{readfield}\ \textit{exp}\ \textit{id} \mid \\
 & & \texttt{plus}\ \textit{exp}\ \textit{exp} \mid \texttt{minus}\ \textit{exp}\ \textit{exp} \mid \\
 & & \texttt{times}\ \textit{exp}\ \textit{exp} \mid \texttt{equal}\ \textit{exp}\ \textit{exp} \mid \texttt{and}\ \textit{exp}\ \textit{exp} \mid \\
 & & \texttt{or}\ \textit{exp}\ \textit{exp} \mid \texttt{less}\ \textit{exp}\ \textit{exp}
\end{array}
$$

Activités complémentaires

- Revoir vos connaissances en compilation et implantation de langages de programmation.
- Découvrir Prolog.
- Découvrir l'analyseur lexico-syntaxique de BOPL écrit en Prolog.
  ♠ fichiers `scanner.ecl` et `parser.ecl` ♣