

MI030 — APS Analyse des programmes et sémantique

© Jacques Malenfant, 2010–2014

avec la participation initiale d'Olena Rogovchenko

Université Pierre et Marie Curie
UFR 919 Ingénierie
Jacques.Malenfant@lip6.fr

Cours 1 Introduction

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.

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.

1 Sémantique et analyse des langages de programmation

- Définition et spécification des langages
- Grammaires et syntaxe
- Différentes formes de sémantiques
- Différentes formes d'analyses sémantiques

2 Le langage BOPL

1 Sémantique et analyse des langages de programmation

- Définition et spécification des langages
- Grammaires et syntaxe
- Différentes formes de sémantiques
- Différentes formes d'analyses sémantiques

2 Le langage BOPL

Langages de programmation I

Définition (Langage 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."
⇒ Le/chat/mange/la/souris.*

Langages de programmation II

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

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 la reconnaissance des phrases du programme, et construit les relations d'imbrication.

Langages 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.

Spécification des langages de programmation I

Définition (Spécification d'un langage 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 :

- définit les *lexèmes* du langage ;
- utilise le plus souvent des *expressions régulières* ;
- $analex : Char^* \rightarrow Lexème^*$.

Spécification des langages de programmation II

Niveau syntaxique :

syntaxe concrète : grammaires indépendantes du contexte
syntaxe abstraite : grammaires abstraites, définissant les arbres de syntaxe abstraite ;
analyse syntaxique : $anasyn : Lexème^* \rightarrow AST$.

Niveau sémantique : pour la signification, on utilise

- les sémantiques discursives,
- les sémantiques opérationnelles classiques,
- les **sémantiques opérationnelles structurales**,
- les sémantiques axiomatiques,

Spécification des langages de programmation III

- **sémantiques dénotationnelles**,
- les sémantiques algébriques,
- ...

Pour les propriétés, on utilise :

- des grammaires attribuées,
- des **analyses dirigées par la syntaxe** (dont l'interprétation abstraite),
- des analyses dirigées par le flot de contrôle,
- ...

1 Sémantique et analyse des langages de programmation

- Définition et spécification des langages
- Grammaires et syntaxe
- Différentes formes de sémantiques
- Différentes formes d'analyses sémantiques

2 Le langage BOPL

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.

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*.

Grammaire concrète et analyse syntaxique 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 syntaxique 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 différencie 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 dites indépendantes du contexte.

Grammaire concrète et analyse syntaxique III

- 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.

Et plus formellement...

Définition

Une grammaire non-contextuelle est 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 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.

Exemple : les expressions arithmétiques simples

Les règles de production suivantes définissent une grammaire pour les expressions arithmétiques simples :

$$\begin{aligned} E &::= E + T \mid E - T \mid T \\ T &::= T * F \mid T / F \mid F \\ F &::= (E) \mid \text{int} \end{aligned}$$

Axiome S : E (le premier non-terminal défini)

Non-terminaux V : $\{E, T, F\}$ (apparaissent en partie gauche)

Terminaux T : $\{ (,), \text{int}, -, +, *, / \}$ (n'apparaissent pas en partie gauche)

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.

Dérivation II

- 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.
- Exemple :

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

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.

Choix du non-terminal à remplacer

- Le choix du non-terminal à remplacer aux différentes étapes de dérivation peut être facilement réglé en appliquant une décision systématique.

Définition (Dérivation à gauche)

Une dérivation à gauche (notée \Rightarrow_g) 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 \Rightarrow_d) 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

Considérons la grammaire :

$$\begin{aligned} S &\rightarrow aABe \\ A &\rightarrow A|bc \mid b \\ B &\rightarrow d \end{aligned}$$

La dérivation à droite de la phrase abbcd e est :

$$S \xRightarrow{d} aABe \xRightarrow{d} aAde \xRightarrow{d} aAbcde \xRightarrow{d} abbcd e$$

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 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 la 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.

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 ;
- 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.

Principaux algorithmes :

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

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

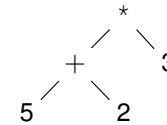
- La syntaxe abstraite ne s'intéresse plus à la représentation du programme dans un fichier, mais bien à son contenu et à sa structure.
- Elle s'intéresse donc au concept derrière les instructions plutôt qu'à leur forme externe : le concept d'instruction d'affectation, de répétition, d'alternative, ...
- 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 I

- La grammaire abstraite des expressions arithmétiques :

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

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



1 Sémantique et analyse des langages de programmation

- Définition et spécification des langages
- Grammaires et syntaxe
- Différentes formes de sémantiques
- Différentes formes d'analyses sémantiques

2 Le langage BOPL

Sémantique discursive

Définition (Sémantique discursive)

Spécification informelle par laquelle la signification des constructions du langage est donnée sous la forme d'explications en langue naturelle.

- 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.

Sémantique opérationnelle classique I

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.

Sémantique opérationnelle classique II

- La sémantique du langage qu'on cherche à définir ne pourra pas être moins ambiguë que la sémantique du langage utilisé pour écrire l'interprète ou vers lequel on traduit.

Une forme particulière : 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.

Sémantique opérationnelle structurale (SOS) I

Définition (Sémantique opérationnelle structurale)

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.

Sémantique opérationnelle structurale (SOS) II

- Les machines abstraites formalisées se définissent par un état (tuples, listes, ..., bien définis) et un ensemble de relations opérant sur ces structures de données.

$$\frac{\Gamma \vdash e_1 \rightarrow v_1 \quad \Gamma \vdash e_2 \rightarrow v_2}{\Gamma \vdash e_1 + e_2 \rightarrow v_1 + v_2}$$

- Relativement facile à comprendre, les SOS fournissent un bon médium de communication avec les utilisateurs et les planteurs.

Sémantique opérationnelle structurelle (SOS) III

- 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.
- 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.

Sémantique opérationnelle structurelle (SOS) IV

- Le grand intérêt de cette approche est d'autoriser les preuves de propriétés par induction structurelle sur les constructions du langage 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).

Sémantique axiomatique II

- 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.

Sémantique axiomatique III

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

$$\{Précondition\} \text{ inst } \{Poscondition\}$$

Sémantique axiomatique IV

- Exemple : sémantique de l'affectation

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

et sur un cas particulier, on pourrait avoir :

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

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

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

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.

Sémantique dénotationnelle II

- 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.

Sémantique dénotationnelle III

- 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.

Sémantique dénotationnelle IV

- Les sémantiques dénotationnelles faisant la correspondance entre de la syntaxe et des objets mathématiques, la présentation des règles de construction met l'emphasis sur ce point en encadrant les formes syntaxiques par des doubles crochets :

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

Sémantique algébrique I

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'emphasis 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.

Sémantique algébrique II

- 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.

1 Sémantique et analyse des langages de programmation

- Définition et spécification des langages
- Grammaires et syntaxe
- Différentes formes de sémantiques
- Différentes formes d'analyses sémantiques

2 Le langage BOPL

Les grammaires attribuées I

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.

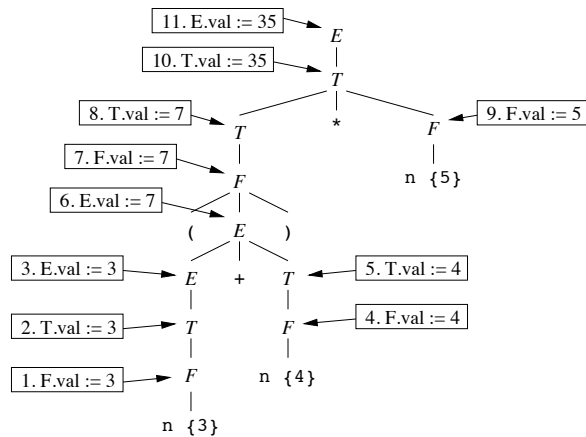
Exemple I : Évaluation d'une expression arithmétique

- 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

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

Exemple III : évaluation des attributs



Analyses dirigées par la syntaxe I

- 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.

Analyses dirigées par la syntaxe II

- 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 I

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 ;

Interprétation abstraite II

- 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 dénotationnelle.

Exemple (classique) d'interprétation abstraite : la règle des signes I

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

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

Exemple (classique) d'interprétation abstraite : la règle des signes II

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

\times_{abs}	+	0	-
+	+	0	-
0	0	0	0
-	-	0	+

- Analyse du signe du résultat : $signe : S \times_{abs} S \rightarrow S$

Les analyses dirigées par la flût de contrôle I

Définition

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 analyses dirigées par la flût de contrôle II

- 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).

1 Sémantique et analyse des langages de programmation

2 Le langage BOPL

- Syntaxe concrète du langage BOPL
- Syntaxe abstraite du langage BOPL

Présentation du langage BOPL

- *Basic Object Programming Language*.
- Dû à Palsberg et Schwartzbach :
[PaSc94] Palsberg, J. et Schwartzbach, 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.

Brève description du langage BOPL II

- 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).
- 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.

Brève description du langage BOPL III

- 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).

Brève description du langage BOPL IV

- 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.

- 1 Sémantique et analyse des langages de programmation
- 2 Le langage BOPL
 - Syntaxe concrète du langage BOPL
 - Syntaxe abstraite du langage BOPL

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 ( Actualist ) |
            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 )
Actualist ::= ε | Actuals
Actuals  ::= Exp | Actuals , Exp

```

Un exemple de programme BOPL I

```

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) ;

```

Un exemple de programme BOPL II

```
    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) ;
```

Un exemple de programme BOPL III

```
writeln(p3)
end
```

1 Sémantique et analyse des langages de programmation

2 Le langage BOPL

- Syntaxe concrète du langage BOPL
- Syntaxe abstraite du langage BOPL

Grammaire abstraite du langage BOPL I

```
program ::= program class* var* inst
class   ::= class id cexp var* method*
cexp    ::= cexp id
var     ::= var cexp id
method  ::= method id var* cexp var* inst
```

Grammaire abstraite du langage BOPL II

```
inst ::= seq inst inst | assign id exp | writefield exp id exp |  
      if exp inst inst | while exp inst | return exp |  
      writeln exp  
exp ::= int | true | false | not exp |  
      nil | self | super | new cexp | instanceof exp exp |  
      id | methodcall exp id exp* | readfield exp id |  
      plus exp exp | minus exp exp |  
      times exp exp | equal exp exp | and exp exp |  
      or exp exp | less exp exp
```

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.