Le premier but de ce livre était de présenter les principaux outils qui permettent de définir la sémantique d'un langage de programmation : la sémantique opérationnelle à petits pas, la sémantique opérationnelle à grands pas et la sémantique dénotationnelle.

Nous avons insisté sur l'unité de but de ces différents outils. Dans les trois cas, il s'agit de définir une relation $\hookrightarrow$ qui relie un programme, une valeur d'entrée et une valeur de sortie. Puisque le but est de définir une relation, il est utile de commencer par s'interroger sur la manière dont le langage mathématique permet de définir des relations.

Il y a également une certaine unité de moyens entre ces outils qui reposent finalement tous les trois sur le théorème du point fixe. Cependant, cette unité de moyens est peut-être plus superficielle qu'il paraît à première vue, car le théorème du point fixe est utilisé d'une manière très différente dans ces trois cas. En servant de fondement à la notion de définition inductive et donc à celle de fermeture réflexive-transitive, le théorème du point fixe joue un rôle majeur dans la définition des sémantiques opérationnelles. En revanche, il joue un rôle moindre dans la définition de la sémantique dénotationnelle, puisqu'il n'est utilisé que pour définir la sémantique de la construction `fix` du langage. La sémantique dénotationnelle d'un langage sans point fixe, comme celle du système T de Gödel — voir l'exercice 5.13 — se définit sans utiliser le théorème du point fixe.

Pour mettre les différences davantage en lumière, on peut insister sur le rôle des dérivations. Pour établir qu'un terme `t` a une valeur `V` en sémantique opérationnelle, il suffit de fournir une dérivation ou une suite de réductions, c'est-à-dire, dans les deux cas, un objet fini. En sémantique dénotationnelle, en revanche, la sémantique d'un terme de la forme `fix`, se définit comme le plus petit point fixe d'une fonction, c'est-à-dire comme une limite. De ce fait, on ne peut pas toujours établir qu'un terme `t` a une valeur `V` en exhibant un objet fini, et il est parfois nécessaire, pour l'établir, de démontrer qu'une suite a une certaine valeur pour limite.

Les sémantiques opérationnelles ont donc un avantage certain sur la sémantique dénotationnelle, car la relation $\hookrightarrow$ y est définie d'une manière beaucoup plus concrète. Le revers de cette médaille est que l'on ne peut définir en sémantique opérationnelle que des relations récursivement énumérables, alors que l'on peut définir des relations quelconques en sémantique dénotationnelle. De ce fait, en sémantique opérationnelle, on ne peut pas compléter la relation $\hookrightarrow$ en ajoutant une valeur $\bot$ pour les termes qui ne terminent pas, car la relation

ainsi obtenue, qui n'est pas récursivement énumérable, ne peut pas être définie inductivement avec des règles effectives. Ajouter une telle valeur ne pose, en revanche, aucun problème en sémantique dénotationnelle.

On voit ici le dilemme que pose l'indécidabilité du problème de l'arrêt : on ne peut pas à la fois compléter la relation $\hookrightarrow$ en ajoutant une valeur $\perp$ pour les termes qui ne terminent pas et définir cette relation inductivement. De ce fait, il faut choisir entre compléter la relation ou la définir inductivement, ce qui mène à une sémantique ou à une autre. Ceux qui ont suivi un cours de logique reconnaîtront ici le même type de distinction qui oppose les jugements de vérité inductivement définis, par l'existence d'une démonstration, à ceux définis par la validité dans un modèle.

Le deuxième but de ce livre était de présenter la sémantique de quelques fonctionnalités des langages de programmation : définitions explicites de fonctions, définitions de fonctions par un point fixe, affectations, enregistrements, objets... Ici encore, puisque le but est de définir des fonctions, il est utile de commencer par s'interroger sur la manière dont le langage mathématique permet de définir des fonctions. De manière générale, la comparaison du langage mathématique et des langages de programmation est souvent une démarche féconde, car le langage mathématique est la chose la plus proche des langages de programmation que l'on connaisse. Cette comparaison montre des convergences entre le langage mathématique et les langages de programmation, mais aussi un certain nombre de divergences.

Le but de l'étude de ces différentes fonctionnalités n'était pas d'être exhaustif, mais de donner des exemples représentatifs. Le point à retenir est que, de même que la zoologie ne consiste pas à étudier successivement toutes les espèces animales, l'étude des langages de programmation ne consiste pas à étudier successivement tous les langages, mais s'organise autour des fonctionnalités que l'on retrouve dans les différents langages.

On pourrait continuer cette étude par la définition de types de données et par les exceptions. L'étude des types de données nous donnerait l'occasion d'utiliser une nouvelle fois le théorème du point fixe, et également une nouvelle fois l'algorithme d'unification de Robinson, dont le filtrage est un cas particulier. En continuant dans cette direction, on pourrait étudier le retour arrière et arriver à Prolog. D'autres points importants que nous avons laissés de coté sont le typage polymorphe des références, les tableaux, les objets impératifs, les modules, le typage des enregistrements et des objets et en particulier la notion de sous-typage, la concurrence...

Le dernier but de ce livre était de présenter un certain nombre d'utilisations de ces outils, en particulier l'écriture d'évaluateurs, d'interpréteurs et de compilateurs, mais aussi de programmes d'inférence de types. Le point à retenir ici est que la structure d'un compilateur est directement dérivée des règles de la sémantique opérationnelle du langage que l'on cherche à compiler. Une manière de poursuivre serait d'étudier les techniques d'implantation des machines abstraites, ce qui nous amènerait à parler, en particulier, de la gestion de la mémoire et du glanage des cellules. Une autre serait d'étudier les systèmes permettant d'analyser les programmes et d'établir de manière interactive ou automatique

que la valeur associée par la sémantique à un programme a une certaine propriété, par exemple que la valeur retournée par un algorithme de tri est bien une liste ordonnée.

Il nous reste une dernière question à discuter, qui est celle de l'utilité de la théorie des langages de programmation, en particulier celle de savoir si le but de cette théorie est de décrire des langages de programmation tels qu'ils sont, ou de proposer de nouveaux langages.

Les astronomes étudient les galaxies telles qu'elles sont et ne sont pas amenés à en construire eux-mêmes. Les chimistes, en revanche, étudient des molécules déjà présentes dans la Nature, et en fabriquent également de nouvelles. On sait que, dans ce second cas, l'ordre dans lequel apparaissent les théories explicatives et les techniques de production est très variable : la transformation de masse en énergie n'a été réalisée que longtemps après la théorie de la relativité, la machine à vapeur, en revanche, a précédé les principes de la thermodynamique.

La théorie des langages de programmation a permis de proposer de nouvelles fonctionnalités comme la liaison statique, l'inférence de types, le typage polymorphe, le glanage de cellules, ... que l'on commence à voir apparaître dans des langages largement diffusés. En revanche, d'autres fonctionnalités, comme les affectations ou les objets, ont été introduites de manière sauvage dans les langages de programmation et la théorie a parfois peiné à suivre le mouvement et à décrire leur sémantique. Cette description a cependant souvent produit de nouvelles propositions en retour, comme les récentes propositions d'extensions de Java avec des types polymorphes.

La théorie des langages de programmation n'a donc ni un rôle exclusif de proposition ni un rôle exclusif de description, et c'est dans ce va-et-vient entre description et proposition qu'elle trouve sa dynamique.