

Sémantique des Langages de Programmation et Compilation

Sémantique des Langages de Programmation

Jean-Claude Fernandez & Yassine Lakhnech

{fernand,lakhnech}@imag.fr

<http://www-verimag.imag.fr/~fernand>

<http://www-verimag.imag.fr/~lakhnech>.

Cours en Master Sciences, Technologies & Santé

Mention Mathématiques, Informatique

M1 majeure Informatique

UFR IMA

Université Joseph Fourier

Equipe pédagogique

Cours

Jean-Claude Fernandez (VERIMAG, av. de Vignate
GIERES 04 56 52 03 79)

TD

Fabienne Carrier (VERIMAG)
Jean-Claude Fernandez (VERIMAG)

<http://www-verimag.imag.fr/~fernand/SLPC>

Objectifs

Etude des théories, méthodes et outils en vue de :

- Définir les langages de programmation (syntaxe, sémantique),
- Traduire efficacement et correctement un programme en vue de l'exécuter (compilation)
- Langages et leur description
- Architecture machine

A la base : traduire un programme écrit dans un langage de programmation vers un programme exécutable par une machine.

Langages

- Langages impératifs e.g. Fortran, Algol-xx, Pascal, C, Ada, Java,... (Concepts : désignation d'emplacements mémoire, Structure de contrôle (PC) expression (PO))
- Langages fonctionnels e.g. ML, CAML, LISP (Concepts : réduction, évaluation de fonctions)
- Langages orienté objet Ada, Java,... (Concepts : objets, classes, types, héritage, polymorphisme, etc)
- Langages logiques Prolog (Concepts : Résolution dans le calcul des prédicats)
- Langages spécialisés VHDL, CSH, HTML, ...
- ...

Langages impératifs

- Structures de données :
 - Types de base (entiers, caractères, etc)
 - Types construits (énumération, produit, union, fonction ou tableaux)
- Expressions
- Structures de contrôle :
 - Affectation
 - itération, conditionnelle, séquence, structure de bloc, appel de fonctions, paramètres

Description des langages

Lexique Les mots du langage (le lexique) sont décrits par des expressions régulières

Syntaxe La syntaxe est décrite à l'aide de grammaire hors-contexte

Sémantique statique (typage) Règles d'inférences ou grammaires attribuées

Sémantique dynamique Règles de dérivation, ensemble de fonctions, ensemble de séquences d'exécution.

ensemble d'équations

Architectures matérielles

Exécuter les programmes

- CISC
- RISC
- VLIW, machines parallèles
- Processeurs spécifiques,
- etc.

Compilation / Interprétation

Compilation Traduit un programme écrit dans un langage L_1 en programme écrit en L_2 (C vers assembleur)
Exécution : assembleurs, chargeur éditeur de lien, lanceur.

Interprétation Combinaison de la traduction et de l'exécution

- Un programme compilé s'exécute plus rapidement, surtout si il y a prise en compte des spécificités matérielles. La compilation (*statique*) peut consacrer du temps à l'analyse de code et à l'optimisation.
- Les programmes interprétés sont généralement plus petits en taille,
- Les programmes interprétés sont plus portables,
- L'interpréteur peut avoir accès à des informations "à l'exécution" (*run time*).

Compilation dynamique : JIT (Just In Time) combine l'interprétation et la compilation.

Plan du cours

Deux thèmes liés :

- La compilation
- Sémantique

Plan prévisionnel :

- Introduction à la compilation : 1 cours
- Sémantique opérationnelle : 3 cours
- Typage : 2 cours
- Génération de code intermédiaire : 1 cours
- Optimisation : 3 cours
- Génération de code : 4 cours
- Hoare : 1 cours
- Techniques avancées de compilation : 2 cours
- Conclusion : 1 cours

References

- [1] A. Aho, R. Sethi and J. Ullman *Compilateurs : Principes, techniques et outils* InterEditions, 1989
- [2] H. R. Nielson and F. Nielson. *Semantics with Applications: An Appetizer*. Springer, March 2007. ISBN 978-1-84628-691-9
- [3] W. Waite and G. Goos. *Compiler Construction* Springer Verlag, 1984
- [4] R. Wilhelm and D. Maurer. *Théorie, construction, génération* Masson 1994

Architecture d'un compilateur

Analyse lexicale : Automates d'états finis, langages réguliers

Analyse syntaxique : Automates à pile, langages hors-contexte

Analyse sémantique : Sémantique des langages

Génération de code intermédiaire : Sémantique des langages

Optimisation de code : Sémantique des langages

Génération de code : Sémantique des langages

Tables des symboles

Analyse lexicale : principe

Entrée : séquence (chaîne) de caractères

Sortie : séquence d'unités (classes) lexicales

1. Analyse la plus longue chaîne correspondant à une classe lexicale
2. Retourne à l'analyseur syntaxique
 - La classe lexicale (le token) : constantes entières ou booléennes, identificateurs, mots clés, opérateurs ou séparateurs.
 - L'élément de cette classe (le lexème)
3. Insère un identificateur dans la table des symboles,
4. Gère les erreurs,
5. Ignore les commentaires.

Analyse lexicale : mise en œuvre

Automate d'états finis déterministe $(Q, \Sigma, q_0, \delta, F)$

Configuration initiale (q_0, w) ,

Configuration finale (q_f, ϵ) , $q_f \in F$

Dérivation $(q, a.w) \longrightarrow (q', w)$ ssi $\delta(q, a) = q'$

Automate " \iff " Expressions régulières : Soit \mathcal{L} un langage. \mathcal{L} est reconnu par un automate d'états finis déterministe si et seulement si il existe une expression régulière qui caractérise \mathcal{L}

Automate " \iff " Grammaire : Soit \mathcal{L} un langage. \mathcal{L} est reconnu par un automate d'états finis déterministe si et seulement si il existe une grammaire qui le génère

Exemple de générateur d'analyseurs : LeX

Analyse syntaxique : principe

Entrée : une séquence d'unités lexicales

Sortie un arbre abstrait, la table des symboles modifiée.

1. Analyse syntaxique du programme,
2. Construction de l'arbre abstrait (AST : Abstract Syntax Tree)
3. Appel de l'analyseur lexical pour avoir une nouvelle unité lexicale.

Analyse syntaxique : Mise en œuvre

Analyseur

- Analyse syntaxique descendante récursive ou prédictive
- Analyse syntaxique ascendante (un automate à pile)

Grammaires hors-contexte et automates à pile

- **Automate pile \iff Grammaires hors-contexte**
- Contrairement aux automates d'états finis, il n'y a pas équivalence entre automates à pile déterministes et automates à piles non-déterministes.

Exemple de générateur d'analyseurs : Yacc, Bison

Analyse syntaxique : Mise en œuvre

Grammaire hors contexte :

$$G = (V_T, V_N, S, P)$$

Langage $\mathcal{L}(G) = \mathcal{L}(S) = \{w \mid S \rightarrow^* w \in V_T^*\}$

Automate à pile :

- Q : ensemble fini d'états, état initial, noté q_0
- Σ : alphabet d'entrée
- Γ : alphabet de pile, élément initial de pile, noté Z_0
- ($F \subseteq Q$ états accepteurs)
- $\delta : Q \times \Gamma^* \times (\Sigma \cup \{\epsilon\}) \longrightarrow 2^{Q \times \Gamma^*}$ est la fonction de transition

Relation de transition

Configurations $(q, \gamma, w) \in Q \times \Gamma^* \times \Sigma^*$

- q : un état
- γ : un mot défini sur l'alphabet de pile
- w : un mot de défini sur l'alphabet d'entrée

Configuration initiale : $C_0 < q_0, Z_0, w >$ ou $< q_0, \epsilon, w >$

Dérivation $(q, \alpha\beta, uw') \longrightarrow (q', \alpha\gamma, w')$ si et seulement si $(q', \gamma) \in \delta(q, \beta, u)$.

Critère d'arrêt : pile vide A partir de la configuration initiale C_0 , où w est le mot à reconnaître, on cherche une configuration $< q, \epsilon, \epsilon >$

Langage accepté $\mathcal{L}(P) = \{w \mid \exists q. C_0 \longrightarrow^* < q, \epsilon, \epsilon >\}$

Il existe un autre critère d'acceptation équivalent : sur état final.

Grammaire et automate à pile : Analyse ascendante non déterministe

A une grammaire hors-contexte $G = (V_T, V_N, Z, P)$ on peut associer un automate à pile reconnaissant le même langage.

$P = (Q, \Sigma, \Gamma, \delta, q, Z_0, F)$ où :

$$Q = \{q, t\}, \quad \Sigma = V_T, \quad Z_0 = \epsilon, \quad F = \emptyset, \quad \Gamma = V,$$

$$\left\{ \begin{array}{l} \delta(q, \epsilon, t) = \{(q, t) \mid t \in V_T\} \\ \delta(q, \alpha_1 \cdots \alpha_n, \epsilon) = \{(q, A) \mid A \longrightarrow \alpha_1 \cdots \alpha_n \in P\}. \\ \delta(q, Z, \epsilon) = \{(f, \epsilon)\} \end{array} \right.$$

Grammaire et automate à pile : Analyse descendante non déterministe

A une grammaire hors-contexte $G = (V_T, V_N, Z, P)$ on peut associer un automate à pile $P = (Q, \Sigma, \Gamma, \delta, q, Z_0, F)$ où :

$$Q = \{q\}, \quad \Sigma = V_T, \quad Z_0 = Z, \quad F = \emptyset, \quad \Gamma = V,$$

$$\begin{cases} \delta(q, t, t) &= \{(q, \epsilon)\} \text{ pour } t \text{ dans } V_T \\ \delta(q, A, \epsilon) &= \{(q, \alpha_n \cdots \alpha_1) \mid A \longrightarrow \alpha_1 \cdots \alpha_n \in P\}. \end{cases}$$

Grammaire et automate à pile :

Analyse déterministe :

Ascendante LR(k), SLR(k), LALR(k)

Descendante LL(K)

Analyse Sémantique : principe et mise en œuvre

Principe

Entrée : AST

Sortie : AST enrichi

- Identification des noms,
- Vérification des types.

Mise en œuvre

- Liaison utilisation définition
- Calcul et vérification des types.

Génération de code : principe et mise en œuvre

Principe

Entrée : AST

Sortie : Code intermédiaire ou code machine cible

Mise en œuvre

- basée sur la sémantique du langage source et la sémantique du langage cible
- fonction de traduction f .
- On montre

$$\text{Sem}_{\text{source}}(P) = \text{Sem}_{\text{cible}}(f(P))$$

Optimisation de code : principe et mise en œuvre

Principe

Entrée sortie : Code intermédiaire

- Critères :
 - Taille du code (taille mémoire),
 - temps d'exécution,
 - consommation d'énergie.

Mise en œuvre

- Analyse flot de données,
- Analyse basée sur des contraintes,
- Interprétation abstraite,
- systèmes de typages.

Sémantique opérationnelle

Deux types de sémantique opérationnelle:

- naturelle (ou à grand pas)
- structurelle (ou à petit pas)

Motivation

Pourquoi étudier la sémantique d'un LP?

La sémantique est essentielle pour :

- comprendre les programmes
- tester et vérifier les programmes
- écrire et comprendre des spécifications
- écrire des compilateurs
- classer les langages de programmation

Pourquoi formaliser la sémantique?

Exemple : Liens statiques vs. liens dynamiques

Program Static_Dynamic

var a ;

proc $p(x)$;

 var a ;

 begin

$a := x + 1; write(x)$

 end;

begin

$a := 0; p(a)$

end;

Exemple : Liens statiques vs. liens dynamiques

```
Program Static_Dynamic
var a;
proc p(x);
    var a;
    begin
        a := x + 1; write(x)
    end;
begin
    a := 0; p(a)
end;
```

Quelle valeur est imprimée?

Exemple : Passage de paramètres

Program value_reference

var a ;

proc $p(x)$;

begin

$a := x + 1; write(a); write(x)$

end;

begin

$a := 2; p(a); write(a)$

end;

Exemple : Passage de paramètres

Program value_reference

var a ;

proc $p(x)$;

begin

$x := x + 1; write(a); write(x)$

end;

begin

$a := 2; p(a); write(a)$

end;

Quelles valeurs sont imprimées?

Exemple : Passage de paramètres

Program value_reference

var a ;

proc $p(x)$;

begin

$x := x + 1; write(a); write(x)$

end;

begin

$a := 2; p(a); write(a)$

end;

2 3 2 , si appel par valeur

3 3 3, si appel par référence

Contenu du cours

- Style de sémantique :
 - Sémantique opérationnelle naturelle
 - Sémantique axiomatique : Logique de Hoare propriétés.
- Langages considérés :
 - impératif
 - fonctionnel

Les définitions inductives

Fermeture Soit E un ensemble, $f : E \times \dots \times E \mapsto E$ une fonction partielle et un sous-ensemble $A \subseteq E$. A est fermé par f ssi $f(A) \subseteq A$.

Définitions inductives Soit E un ensemble, une définition inductive sur E est une famille de règles (fonctions partielles) définissant le plus petit ensemble A de E , fermé par ces règles.

Exemples de définitions inductives d'un ensemble

L'ensemble des entiers naturels

- 0,
- si $x \mapsto s(x)$.

Les entiers pairs

- 0,
- $x \mapsto x + 2$

Les palindromes Soit $\Sigma = \{a, b\}$.

- ϵ
- $u \mapsto a.u.a$
- $u \mapsto b.u.b$

Notation

$$t = f(x_1, \dots, x_n)$$

est noté

$$\frac{x_1 \cdots x_n}{t}$$

Outils mathématiques

Soit (E, \leq) un ensemble ordonné.

Limite d'une suite croissante $\{u_i\}_{i \in I}$ $l = \lim_{i \in I} u_i$ si

- l est un majorant : $\forall i \in I \cdot u_i \leq l$
- c'est le plus petit : $\forall l' \cdot \forall i \cdot (u_i \leq l' \implies l \leq l')$

Relation faiblement complète toute suite croissante a une limite

Relation fortement complète tout sous-ensemble de E a une borne supérieure (le plus petit des majorants).

Fonctions croissantes continues

Fonction croissante

$$x \leq y \implies f(x) \leq f(y)$$

Fonction continue si pour toute suite croissante $\{u_i\}_{i \in I}$

$$\lim_{i \in I} (f(u_i)) = f(\lim_{i \in I} (u_i))$$

Théorème du point fixe

Premier théorème Si la relation d'ordre est faiblement complète, si E a un plus petit élément m , alors toute fonction continue $f : E \mapsto E$ a un plus petit point fixe

$$\lim_{i \in I} (f^i(m))$$

Second théorème Si la relation d'ordre est fortement complète, toute fonction croissante $f : E \mapsto E$ a un plus petit point fixe

$$\inf\{C \mid f(C) \leq C\}$$

Application aux définitions inductives

Proposition 1 Soit E un ensemble, f_1, f_2, \dots des règles. Il existe un plus petit ensemble de E fermé par ces règles (application du second théorème du point fixe).

Proposition 2 Ce plus petit sous-ensemble est

$$\bigcup_k F^k(\emptyset)$$

où

$$F(X) = \{x \in E \mid \exists i. z_1 \cdots z_{n_i}. x = f_i(z_1, \dots, z_{n_i})\}$$

Récurrance structurelle

Les définitions inductives donnent un moyen de faire des démonstration. Si une propriété est *héréditaire* ($P(x_i) \implies P(f(x_1, \dots, x_n))$), alors elle est vérifiée par tous les éléments de A .

Une manière de le démontrer est d'utiliser le second théorème, l'autre est de vérifier que tout élément de $F^k(\emptyset)$ vérifie la propriété.

Dérivations

On a

$$x \in A \text{ ssi } \exists k \cdot x \in F^k(\emptyset)$$

On peut montrer que $x \in A$ ssi

- il existe un arbre dont chaque nœud est étiqueté par un élément x de A et ses enfants par des éléments y_i tels que il existe une règle $x = f(x_1, \dots, x_n)$ ou
- chaque nœud est étiqueté par une règle et vérifiant quelque chose d'analogue.

Le langage **While**

x : variable

S : commande

a : expression arithmétique

b : expression booléenne

$$S ::= x := a \mid \text{skip} \mid S_1; S_2 \mid$$
$$\text{if } b \text{ then } S_1 \text{ else } S_2$$
$$\text{while } b \text{ do } S \text{ od}$$

Catégories syntaxiques 1

- Chiffres

$$n \in \mathbf{Num} = \{0, \dots, 9\}^+$$

- Variables

$$x \in \mathbf{Var}$$

- Expressions arithmétiques

$$a \in \mathbf{Aexp}$$

$$a := n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2$$

Catégories syntaxiques 2

- Expressions booléennes

$$b \in \mathbf{Bexp}$$
$$b ::= \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$$

- Commandes

$$S \in \mathbf{Stm}$$
$$S ::= x := a \mid \text{skip} \mid S_1; S_2 \mid$$
$$\text{if } b \text{ then } S_1 \text{ else } S_2$$
$$\text{while } b \text{ do } S \text{ od}$$

Syntaxe concrète vs. syntaxe abstraite

- Le terme $S_1; S_2$ représente l'arbre de racine $;$, fils gauche l'arbre de S_1 et fils droite celui de S_2 .
- Nous utilisons des parenthèses pour lever des ambiguïtés.
Exemple :

$S_1; (S_2; S_3)$ et $(S_1; S_2); S_3$

Domaines sémantiques

Entiers relatifs : \mathbb{Z}

Booléens : \mathbb{B}

États :

$$\text{State} = \text{Var} \rightarrow \mathbb{Z}$$

Soit $v \in \mathbb{Z}$. Alors, $\sigma[y \mapsto v]$ dénote l'état σ' tel que:

$$\sigma'(x) = \begin{cases} \sigma(x) & \text{si } x \neq y \\ v & \text{sinon} \end{cases}$$

Fonctions sémantiques 1

- Chiffres : entiers relatifs

$$\mathcal{N} : \mathbf{Num} \rightarrow \mathbb{Z}$$

- Expressions arithmétiques : dans chaque état une valeur dans \mathbb{Z}

$$\mathcal{A} : \mathbf{Aexp} \rightarrow (\mathbf{State} \rightarrow \mathbb{Z})$$

Fonctions sémantiques 2

- Expressions booléennes : dans chaque état une valeur dans \mathbb{B}

$$\mathcal{B} : \mathbf{Bexp} \rightarrow (\mathbf{State} \rightarrow \mathbb{B})$$

- Commandes :

$$\mathcal{S} : \mathbf{Stm} \rightarrow (\mathbf{State} \xrightarrow{\text{part.}} \mathbf{State})$$

Sémantique des expressions arithmétiques

$$\mathcal{N}(n_1 \cdots n_k) = \sum_{i=1}^k n_i \cdot 10^{k-i}$$

$$\mathcal{A}[n]\sigma = \mathcal{N}[n]$$

$$\mathcal{A}[x]\sigma = \sigma(x)$$

$$\mathcal{A}[a_1 + a_2]\sigma = \mathcal{A}[a_1]\sigma +_I \mathcal{A}[a_2]\sigma$$

$$\mathcal{A}[a_1 * a_2]\sigma = \mathcal{A}[a_1]\sigma *_I \mathcal{A}[a_2]\sigma$$

$$\mathcal{A}[a_1 - a_2]\sigma = \mathcal{A}[a_1]\sigma -_I \mathcal{A}[a_2]\sigma$$

La sémantique des expressions arithmétiques est définie inductivement sur la structure des expressions. C'est une sémantique compositionnelle.

Sémantique des expressions booléennes

Exercice : Définir la sémantique des expressions booléennes.

Différents styles de sémantique

- La sémantique opérationnelle nous décrit comment un programme est exécuté. Elle nous sert par exemple à écrire un générateur de code.
- La sémantique axiomatique nous permet de prouver des propriétés de nos programmes.
- La sémantique dénotationnelle nous permettra de décrire l'effet d'un programme sur un état sans dire comment le programme est exécuté.

Un autre aspect important est la *compositionnalité*. La sémantique d'un programme composé est une fonction de la sémantique des composants.

Sémantique opérationnelle

Une sémantique opérationnelle définit un système de transition.

Un système de transition est donné par:

$$(\Gamma, T, \rightarrow) \text{ où}$$

- Γ est l'ensemble de configurations.
- $T \subseteq \Gamma$ est l'ensemble des configurations finales.
- $\rightarrow \subseteq \Gamma \times \Gamma$ est la relation de transition.

Sémantique naturelle 1

Idée : Décrire comment le résultat de l'exécution d'un programme est obtenu.

Sémantique décrite par un système d'inférence : axiomes et règles.

$$(x := a, \sigma) \rightarrow \sigma[x \mapsto \mathcal{A}[a]\sigma]$$

$$(\text{skip}, \sigma) \rightarrow \sigma$$

$$\frac{(S_1, \sigma) \rightarrow \sigma', \quad (S_2, \sigma') \rightarrow \sigma''}{(S_1; S_2, \sigma) \rightarrow \sigma''}$$

Sémantique naturelle 2

Si $\mathcal{B}[b]\sigma = \mathbf{tt}$ alors

$$\frac{(S_1, \sigma) \rightarrow \sigma'}{(\text{if } b \text{ then } S_1 \text{ else } S_2, \sigma) \rightarrow \sigma'}$$

Si $\mathcal{B}[b]\sigma = \mathbf{ff}$ alors

$$\frac{(S_2, \sigma) \rightarrow \sigma'}{(\text{if } b \text{ then } S_1 \text{ else } S_2, \sigma) \rightarrow \sigma'}$$

Si $\mathcal{B}[b]\sigma = \mathbf{tt}$ alors

$$\frac{(S, \sigma) \rightarrow \sigma', \quad (\text{while } b \text{ do } S \text{ od}, \sigma') \rightarrow \sigma''}{(\text{while } b \text{ do } S \text{ od}, \sigma) \rightarrow \sigma''}$$

Sémantique naturelle 3

Si $\mathcal{B}[b]\sigma = \mathbf{tt}$ alors

$$\frac{(S, \sigma) \rightarrow \sigma', \quad (\text{while } b \text{ do } S \text{ od}, \sigma') \rightarrow \sigma''}{(\text{while } b \text{ do } S \text{ od}, \sigma) \rightarrow \sigma''}$$

Si $\mathcal{B}[b]\sigma = \mathbf{ff}$ alors

$$(\text{while } b \text{ do } S \text{ od}, \sigma) \rightarrow \sigma$$

Arbre de dérivation

- Les feuilles sont des axiomes
- Les nœuds sont des règles de la sémantique.

Exemple : Calculons la sémantique de:

1. $x := 2; \text{while } x > 0 \text{ do } x := x - 1 \text{ od}$
2. $x := 2; \text{while } x > 0 \text{ do } x := x + 1 \text{ od}$

La sémantique naturelle est déterministe

Théorème Pour toute commande $S \in \mathbf{Stm}$, pour tout états σ, σ' et σ'' :

1. Si $(S, \sigma) \rightarrow \sigma'$ et $(S, \sigma) \rightarrow \sigma''$ alors $\sigma' = \sigma''$.
2. Si $(S, \sigma) \rightarrow \sigma'$ alors il n'existe pas d'arbre de dérivation infini.

Preuve Preuve par induction sur la structure de l'arbre de dérivation. □

La fonction sémantique \mathcal{S}_{ns}

$$\mathcal{S}_{ns}[S]\sigma = \begin{cases} \sigma' & ; \text{ Si } (S, \sigma) \rightarrow \sigma' \\ \text{undef} & ; \text{ sinon} \end{cases}$$

Blocs et procédures

Blocs et déclarations de variables

$S \in \mathbf{Stm}$

$S ::= x := a \mid \text{skip} \mid S_1; S_2 \mid$

$\text{if } b \text{ then } S_1 \text{ else } S_2$

$\text{while } b \text{ do } S \text{ od} \mid \text{begin } D_V; S \text{ end}$

La catégorie syntaxique \mathbf{Dec}_V

$D_V ::= \text{var } x := a; D_V \mid \epsilon$

Exemple

```
begin  var  $y := 1$ ;  
      ( $x := 1$ ;  
      begin var  $x := 2$ ;  $y := x + 1$  end  
       $x := y + x$ )  
end
```

Exemple

```
begin  var  $y := 1$ ;  
      ( $x := 1$ ;  
      begin var  $x := 2$ ;  $y := x + 1$  end  
       $x := y + x$ )  
end
```

Questions :

1. Les déclarations sont-elles déjà actives pour les instructions?
2. Quel ordre choisir pour les déclarations?
3. Comment restaurer l'état?

Sémantique opérationnelle naturelle

Notation :

- $DV(D_V)$ dénote l'ensemble des variables déclarées dans D_V .
- $\sigma'[X \mapsto \sigma] = \lambda x \cdot \text{if } x \in X \text{ then } \sigma(x) \text{ else } \sigma'(x)$.

Pour définir la sémantique, on définit un système de transitions pour les déclarations et un système pour les commandes.

Déclarations :

Configurations de la forme (D_v, σ) ou σ .

$$\frac{(D_V, \sigma[x \mapsto \mathcal{A}[a]\sigma]) \rightarrow_D \sigma'}{(\text{var } x := a; D_V, \sigma) \rightarrow_D \sigma'}$$

$$(\epsilon, \sigma) \rightarrow_D \sigma$$

Règle de transition pour les blocs

$$\frac{(D_V, \sigma) \rightarrow_D \sigma' \quad (S, \sigma') \rightarrow \sigma''}{(\text{begin } D_V; S \text{ end}, \sigma) \rightarrow \sigma''}$$

Règle de transition pour les blocs

$$\frac{(D_V, \sigma) \rightarrow_D \sigma' \quad (S, \sigma') \rightarrow \sigma''}{(\text{begin } D_V; S \text{ end}, \sigma) \rightarrow \sigma'' [\text{DV}(D_V) \mapsto \sigma]}$$

Règle de transition pour les blocs

$$\frac{(D_V, \sigma) \rightarrow_D \sigma' \quad (S, \sigma') \rightarrow \sigma''}{(\text{begin } D_V; S \text{ end}, \sigma) \rightarrow \sigma'' [\text{DV}(D_V) \mapsto \sigma]}$$

Règle pour la composition séquentielle

$$\frac{(S_1, \sigma) \rightarrow \sigma' \quad (S_2, \sigma') \rightarrow \sigma''}{(S_1; S_2, \sigma) \rightarrow \sigma''}$$

Procédure

$S \in \mathbf{Stm}$

$S ::= x := a \mid \text{skip} \mid S_1; S_2 \mid$

$\text{if } b \text{ then } S_1 \text{ else } S_2$

$\text{while } b \text{ do } S \text{ od} \mid \text{begin } D_V \ D_P; S \text{ end} \mid \text{call } p$

$D_V ::= \text{var } x := a; D_V \mid \epsilon$

$D_P ::= \text{proc } p \text{ is } S; D_P \mid \epsilon$

Les déclarations de procédures forment la catégorie syntaxique

\mathbf{Dec}_P .

Exemple

```
begin var  $x := 0$ ;  
      proc  $p$  is  $x := x * 2$ ;  
      proc  $q$  is call  $p$ ;  
      begin var  $x := 5$ ;  
            proc  $p$  is  $x := x + 1$ ;  
            call  $q$ ;  $y := x$ ;  
      end;  
end
```

Exemple 2

Liens dynamiques pour les variables et les procédures.

```
begin var  $x := 0$ ;  
      proc  $p$  is  $x := x * 2$ ;  
      proc  $q$  is call  $p$ ;  
      begin var  $x := 5$ ;  
            proc  $p$  is  $x := x + 1$ ;  
            call  $q$ ;  $y := x$ ;  
      end;  
end
```

Exemple 2

Liens dynamiques pour les variables et les procédures.

```
begin var  $x := 0$ ;  
      proc  $p$  is  $x := x * 2$ ;  
      proc  $q$  is call  $p$ ;  
      begin var  $x := 5$ ;  
            proc  $p$  is  $x := x + 1$ ;  
            call  $p$ ;  $y := x$ ;  
      end;  
end
```

Exemple 2

Liens dynamiques pour les variables et les procédures.

```
begin var  $x := 0$ ;  
      proc  $p$  is  $x := x * 2$ ;  
      proc  $q$  is call  $p$ ;  
      begin var  $x := 5$ ;  
            proc  $p$  is  $x := x + 1$ ;  
             $x := x + 1; y := x$ ;  
      end;  
end
```

Exemple 3

Liens dynamiques pour les variables et liens statiques pour les procédures.

```
begin var  $x := 0$ ;  
      proc  $p$  is  $x := x * 2$ ;  
      proc  $q$  is call  $p$ ;  
      begin var  $x := 5$ ;  
            proc  $p$  is  $x := x + 1$ ;  
            call  $q$ ;  $y := x$ ;  
      end;  
end
```

Exemple 3

Liens dynamiques pour les variables et liens statiques pour les procédures.

```
begin var  $x := 0$ ;  
      proc  $p$  is  $x := x * 2$ ;  
      proc  $q$  is call  $p$ ;  
      begin var  $x := 5$ ;  
            proc  $p$  is  $x := x + 1$ ;  
            call  $p$ ;  $y := x$ ;  
      end;  
end
```

Exemple 3

Liens dynamiques pour les variables et liens statiques pour les procédures.

```
begin var  $x := 0$ ;  
      proc  $p$  is  $x := x * 2$ ;  
      proc  $q$  is call  $p$ ;  
      begin var  $x := 5$ ;  
            proc  $p$  is  $x := x + 1$ ;  
             $x := x * 2; y := x$ ;  
      end;  
end
```

Exemple 4

Liens statiques pour les variables et les procédures.

```
begin var  $x := 0$ ;  
      proc  $p$  is  $x := x * 2$ ;  
      proc  $q$  is call  $p$ ;  
      begin var  $x := 5$ ;  
            proc  $p$  is  $x := x + 1$ ;  
            call  $q$ ;  $y := x$ ;  
      end;  
end
```

Exemple 4

Liens statiques pour les variables et les procédures.

```
begin var  $x := 0$ ;  
      proc  $p$  is  $x := x * 2$ ;  
      proc  $q$  is call  $p$ ;  
      begin var  $x := 5$ ;  
            proc  $p$  is  $x := x + 1$ ;  
            call  $p$ ;  $y := x$ ;  
      end;  
end
```

Exemple 4

Liens statiques pour les variables et les procédures.

```
begin var  $x := 0$ ;  
      proc  $p$  is  $x := x * 2$ ;  
      proc  $q$  is call  $p$ ;  
      begin var  $x := 5$ ;  
            proc  $p$  is  $x := x + 1$ ;  
             $x := x * 2$ ;  $y := x$ ;  
      end;  
end
```

Sémantique : liens dynamiques

Un état associe une valeur, un entier, à une variable.

Un *environnement* associe une valeur à une procédure.

$$\mathbf{Env}_P = \mathbf{Pname} \xrightarrow{\text{part.}} \mathbf{Stm}$$

Les configurations : $(\mathbf{Env}_P \times \mathbf{Stm} \times \mathbf{State}) \cup \mathbf{State}$.

Règles de transitions

$$\frac{(D_V, \sigma) \rightarrow_D \sigma' \quad (\text{upd}(\text{env}, D_P), S, \sigma') \rightarrow \sigma''}{(\text{env}, \text{begin } D_V \ D_P; S \ \text{end}, \sigma) \rightarrow \sigma'' [\text{DV}(D_V) \mapsto \sigma]}$$

où

- $\text{upd}(\text{env}, \epsilon) = \text{env}$ **et**
- $\text{upd}(\text{env}, \text{proc } p \text{ is } S; D_P) = \text{upd}(\text{env}[p \mapsto S], D_P)$.

$$\frac{(\text{env}, \text{env}(p), \sigma) \rightarrow \sigma'}{(\text{env}, \text{call } p, \sigma) \rightarrow \sigma'}$$

Règle pour la composition séquentielle

$$\frac{(env, S_1, \sigma) \rightarrow \sigma' \quad (env, S_2, \sigma') \rightarrow \sigma''}{(env, S_1; S_2, \sigma) \rightarrow \sigma''}$$

Sémantique : liens statiques pour les procédures

$$\mathbf{Env}_P = \mathbf{Pname} \xrightarrow{\text{part.}} \mathbf{Stm} \times \mathbf{Env}_P$$

Les configurations : $(\mathbf{Env}_P \times \mathbf{Stm} \times \mathbf{State}) \cup \mathbf{State}$.

- $\text{upd}(\text{env}, \epsilon) = \text{env}$ et
- $\text{upd}(\text{env}, \text{proc } p \text{ is } S; D_P) = \text{upd}(\text{env}[p \mapsto (S, \text{env})], D_P)$.

```
begin  var  x := 2;
       proc p is x := 0;
       proc q is begin x := 1; (proc p is call p); call p end;
       call q
end
```

Sémantique : liens statiques pour les procédures

$$\mathbf{Env}_P = \mathbf{Pname} \xrightarrow{\text{part.}} \mathbf{Stm} \times \mathbf{Env}_P$$

Les configurations : $(\mathbf{Env}_P \times \mathbf{Stm} \times \mathbf{State}) \cup \mathbf{State}$.

- $\text{upd}(env, \epsilon) = env$ et
- $\text{upd}(env, \text{proc } p \text{ is } S; D_P) = \text{upd}(env[p \mapsto (S, env)], D_P)$.

```
begin  var  x := 2;
       proc p is x := 0;
       proc q is begin x := 1; (proc p is call p); call p end;
       call q
end
```

Sémantique : liens statiques pour les procédures

$$\mathbf{Env}_P = \mathbf{Pname} \xrightarrow{\text{part.}} \mathbf{Stm} \times \mathbf{Env}_P$$

Les configurations : $(\mathbf{Env}_P \times \mathbf{Stm} \times \mathbf{State}) \cup \mathbf{State}$.

- $\text{upd}(\text{env}, \epsilon) = \text{env}$ et
- $\text{upd}(\text{env}, \text{proc } p \text{ is } S; D_P) = \text{upd}(\text{env}[p \mapsto (S, \text{env})], D_P)$.

```
begin  var  x := 2;
        proc p is x := 0;
        proc q is begin x := 1; (proc p is call p); call p end;
        call q
end
```

Règles de transitions

Deux alternatives :

$$[\text{call}] \quad \frac{(env', S, \sigma) \rightarrow \sigma'}{(env, \text{call } p, \sigma) \rightarrow \sigma'} \quad \text{où } env(p) = (S, env').$$

$$[\text{call}_{rec}] \quad \frac{(env'[p \mapsto (S, env')], S, \sigma) \rightarrow \sigma'}{(env, \text{call } p, \sigma) \rightarrow \sigma'} \quad \text{où } env(p) = (S, env').$$

Règles de transitions

Deux alternatives :

$$[\text{call}] \quad \frac{(env', S, \sigma) \rightarrow \sigma'}{(env, \text{call } p, \sigma) \rightarrow \sigma'} \quad \text{où } env(p) = (S, env').$$

$$[\text{call}_{rec}] \quad \frac{(env'[p \mapsto (S, env')], S, \sigma) \rightarrow \sigma'}{(env, \text{call } p, \sigma) \rightarrow \sigma'} \quad \text{où } env(p) = (S, env').$$

Règle pour la composition séquentielle

$$\frac{(env, S_1, \sigma) \rightarrow \sigma' \quad (env, S_2, \sigma') \rightarrow \sigma''}{(env, S_1; S_2, \sigma) \rightarrow \sigma''}$$

Sémantique : liens statiques

On remplace l'état par une table des symboles et la mémoire :

- une table des symboles associe à une variable (un identificateur) une adresse dans la mémoire.
- La mémoire associe à une adresse une valeur.

Table des symboles : environnement des variables :

$$\mathbf{Env}_V = \mathbf{Var} \xrightarrow{\text{part.}} \mathbf{Loc} = \mathbb{Z}$$

Mémoire : $\mathbf{Store} = \mathbf{Loc} \xrightarrow{\text{part.}} \mathbb{Z}$

On dénote par $\text{new}(sto)$ le plus petit entier n tel que $sto(n)$ n'est pas défini.

Intuition : l'état correspond à $sto \circ env_V$.

Configurations

- Déclarations de variables :
 $(\mathbf{Dec}_V \times Env_V \times \mathbf{Store}) \cup (Env_V \times \mathbf{Store})$.
- $\mathbf{Env}_P = \mathbf{Pname} \xrightarrow{part.} \mathbf{Stm} \times \mathbf{Env}_V \times \mathbf{Env}_P$.
 - $upd(env_V, env_P, \epsilon) = env_P$ et
 - $upd(env_V, env_P, \text{proc } p \text{ is } S; D_P) =$
 $upd(env_V, env_P[p \mapsto (S, env_V, env_P)], D_P)$.
- Commandes : $(\mathbf{Env}_V \times \mathbf{Env}_P \times \mathbf{Stm} \times \mathbf{Store}) \cup (\mathbf{Env}_V \times \mathbf{Store})$.

Règles sémantiques pour les déclarations de variables

$$\frac{(D_V, env_V[x \mapsto \text{new}(sto)], sto[\text{new}(sto) \mapsto v]) \rightarrow_D (env'_V, sto')}{(\text{var } x := a; D_V, env_V, sto) \rightarrow_D (env'_V, sto')}$$

où $v = \mathcal{A}[a](sto \circ env_V)$.

$$(\epsilon, env_V, sto) \rightarrow_D (env_V, sto)$$

Règles sémantiques pour les commandes

$$(env_V, env_P, x := a, sto) \rightarrow (env_V, sto[env_V(x) \mapsto \mathcal{A}[a](sto \circ env_V)])$$

$$[\text{call}] \quad \frac{(env'_V, env'_P, S, sto) \rightarrow (env'_V, sto')}{(env_V, env_P, \text{call } p, sto) \rightarrow (env_V, sto')}.$$

$$[\text{call}_{rec}] \quad \frac{(env'_V, env'_P[p \mapsto (S, env'_V, env'_P)], S, sto) \rightarrow (env'_V, sto')}{(env_V, env_P, \text{call } p, sto) \rightarrow (env_V, sto')}$$

où $env_P(p) = (S, env'_V, env'_P)$.

Règles sémantiques pour les commandes

$$(env_V, env_P, x := a, sto) \rightarrow (env_V, sto[env_V(x) \mapsto \mathcal{A}[a](sto \circ env_V)])$$

$$[\text{call}] \quad \frac{(env'_V, env'_P, S, sto) \rightarrow (env'_V, sto')}{(env_V, env_P, \text{call } p, sto) \rightarrow (env_V, sto')}.$$

$$[\text{call}_{rec}] \quad \frac{(env'_V, env'_P[p \mapsto (S, env'_V, env'_P)], S, sto) \rightarrow (env'_V, sto')}{(env_V, env_P, \text{call } p, sto) \rightarrow (env_V, sto')}$$

où $env_P(p) = (S, env'_V, env'_P)$.

Règle pour la composition séquentielle

$$\frac{(env_V, env_P, S_1, \sigma) \rightarrow \sigma' \quad (env_V, env_P, S_2, \sigma') \rightarrow \sigma''}{(env_V, env_P, S_1; S_2, \sigma) \rightarrow \sigma''}$$

Génération de code correcte

Nous allons voir :

- comment on définit une sémantique opérationnelle d'une machine abstraite : une machine à pile.
- comment on peut spécifier un générateur de code pour le langage **While** par induction sur la structure des programmes.
- comment on peut utiliser la sémantique opérationnelle du langage et de la machine abstraite pour montrer que la spécification du générateur de code est correcte.

La machine abstraite AM

La machine AM est défini par un système de transition dont les configurations sont des triplets :

- Une liste $instr_1, \dots, instr_n$ d'instructions. C'est le code qui reste à exécuter.
- Une pile qui est utilisée pour évaluer les expressions.
- La mémoire de la machine qui est décrite par un état; donc une fonction des variables vers \mathbb{Z} .

Nous allons définir l'ensemble des instructions et la relation de transition

$$(c, p, m) \triangleright (c'p', m')$$

La machine AM n'a donc ni registres et ni accumulateur. C'est le sommet de la pile qui joue le rôle d'accumulateur et le reste de la pile celui de registres.

Les instructions

Instructions	Effet
<code>push-n, True, False</code>	empiler la constante n, tt, ff
<code>fetch(x)</code> <code>store(x)</code>	empiler la valeur de x dans l'état actuel dépiler le sommet de la pile et l'affecter à x
<code>add</code> <code>sub, mult, and, le, equal, neg</code> <code>branch(c_1, c_2)</code> <code>loop(c_1, c_2)</code> <code>noop</code>	remplacer les deux plus hauts éléments de la pile par leur somme similaire si le sommet est tt exécuter c_1 s'il est ff alors c_2 , sinon blocage exécuter c_1 , si le sommet de la pile est tt , exécuter c_2 suivie de <code>loop(c_1, c_2)</code> si c'est ff s'arreter skip

La relation de transition-sémantique des instructions

On définit les programmes cibles comme des mots sur l'alphabet des d'instructions. L'ensemble des programmes cibles est dénoté **Code**.

Une configuration de AM est donc (c, p, m) où c est un programme cible, p est le contenu de la pile qui est un mot sur $\mathbb{Z} \cup \mathbb{B}$ et m est un état dans **State**.

La relation \triangleright est inductivement définie par :

$$\begin{array}{ll} (\text{push-n} \cdot c, p, m) & \triangleright (c, n \cdot p, m) \\ (\text{True} \cdot c, p, m) & \triangleright (c, \mathbf{tt} \cdot p, m) \\ (\text{False} \cdot c, p, m) & \triangleright (c, \mathbf{ff} \cdot p, m) \\ (\text{fetch}(x) \cdot c, p, m) & \triangleright (c, m(x) \cdot p, m) \\ (\text{store}(x) \cdot c, v \cdot p, m) & \triangleright (c, p, m[x \mapsto v]) \quad \text{si } v \in \mathbb{Z} \\ (\text{add} \cdot c, v_1 \cdot v_2 \cdot p, m) & \triangleright (c, (v_1 + v_2) \cdot p, m) \quad \text{si } v_1, v_2 \in \mathbb{Z} \\ (\text{sub} \cdot c, v_1 \cdot v_2 \cdot p, m) & \triangleright (c, (v_1 - v_2) \cdot p, m) \quad \text{si } v_1, v_2 \in \mathbb{Z} \end{array}$$

La relation de transition-sémantique des instructions

$$\begin{array}{lll} (\text{mult} \cdot c, v_1 \cdot v_2 \cdot p, m) & \triangleright (c, (v_1 * v_2) \cdot p, m) & \text{si } v_1, v_2 \in \mathbb{Z} \\ (\text{le} \cdot c, v_1 \cdot v_2 \cdot p, m) & \triangleright (c, (v_1 \leq v_2) \cdot p, m) & \text{si } v_1, v_2 \in \mathbb{Z} \\ (\text{equal} \cdot c, v_1 \cdot v_2 \cdot p, m) & \triangleright (c, (v_1 = v_2) \cdot p, m) & \text{si } v_1, v_2 \in \mathbb{Z} \\ (\text{and} \cdot c, b_1 \cdot b_2 \cdot p, m) & \triangleright (c, (b_1 \wedge b_2) \cdot p, m) & \text{si } b_1, b_2 \in \mathbb{B} \\ (\text{neg} \cdot c, b \cdot p, m) & \triangleright (c, (\neg b) \cdot p, m) & \text{si } b \in \mathbb{B} \\ (\text{branch}(c_1, c_2) \cdot c, \text{tt} \cdot p, m) & \triangleright (c_1 \cdot c, p, m) & \\ (\text{branch}(c_1, c_2) \cdot c, \text{ff} \cdot p, m) & \triangleright (c_2 \cdot c, p, m) & \\ (\text{loop}(c_1, c_2) \cdot c, p, m) & \triangleright & \\ & (c_1 \cdot \text{branch}(c_2 \cdot \text{loop}(c_1, c_2), \text{noop}) \cdot c, p, m) & \\ (\text{noop} \cdot c, p, m) & \triangleright (c, p, m) & \end{array}$$

Exemple 1

$x := 2$ se traduit par : $\text{push-2} \cdot \text{store}(x)$

$x := 2; y := x + 4$

se traduit par :

$\text{push-2} \cdot \text{store}(x) \cdot \text{fetch}(x) \cdot \text{push-4} \cdot \text{add} \cdot \text{store}(y)$

Exécution

$(\text{push-2} \cdot \text{store}(x) \cdot \text{fetch}(x) \cdot \text{push-4} \cdot \text{add} \cdot \text{store}(y), \epsilon, \sigma) \triangleright$

$(\text{store}(x) \cdot \text{fetch}(x) \cdot \text{push-4} \cdot \text{add} \cdot \text{store}(y), 2, \sigma) \triangleright$

$(\text{fetch}(x) \cdot \text{push-4} \cdot \text{add} \cdot \text{store}(y), \epsilon, \sigma_2) \triangleright$

$(\text{push-4} \cdot \text{add} \cdot \text{store}(y), 2, \sigma_2) \triangleright$

$(\text{add} \cdot \text{store}(y), 4 \cdot 2, \sigma_2) \triangleright$

$(\text{store}(y), 6, \sigma_2) \triangleright (\epsilon, \epsilon, \sigma_{2ac})$

Exemple 2

while $1 \leq x$ do $x := x - 1$ od

se traduit par :

$\text{loop}(\text{fetch}(x) \cdot \text{push-1} \cdot le, \text{push-1} \cdot \text{fetch}(x) \cdot \text{sub} \cdot \text{store}(x))$

Exemple 3 : A quoi correspond le programme

push-0 · store(z) · fetch(x) · store(r),
loop(fetch(r) · fetch(y) · \leq ,
 fetch(y) · fetch(r) · sub · store(r) · push-1 · fetch(z) · add · store(z))

Exemple 3 : A quoi correspond le programme

push-0 · store(z) · fetch(x) · store(r),
loop(fetch(r) · fetch(y) · \leq ,
 fetch(y) · fetch(r) · sub · store(r) · push-1 · fetch(z) · add · store(z))

Réponse

Exemple 3 : A quoi correspond le programme

push-0 · store(z) · fetch(x) · store(r),
loop(fetch(r) · fetch(y) · \leq ,
 fetch(y) · fetch(r) · sub · store(r) · push-1 · fetch(z) · add · store(z))
 $z := 0$;
 $r := x$;
while $y \leq r$ do $r := r - y$; $z := z + 1$ od

Quelques propriétés de AM

- La relation de transition \triangleright est déterministe :

$$(c, p, m) \triangleright (c_1, p_1, m_1) \wedge (c, p, m) \triangleright (c_2, p_2, m_2) \Rightarrow \\ (c_1, p_1, m_1) = (c_2, p_2, m_2)$$

Démonstration par récurrence sur la longueur de c .
On en déduit qu'à partir d'une configuration, il y a au plus une séquence d'exécution.

- Extensibilité du code et de la pile :

$$(c_1, p_1, m_1) \triangleright (c_2, p_2, m_2) \Rightarrow (c_1 \cdot c, p_1 \cdot p, m_1) \triangleright (c_2 \cdot c, p_2 \cdot p, m_2)$$

- Composabilité du code :

Si $(c_1 \cdot c_2, p, m) \triangleright^k (\epsilon, p_2, m_2)$ alors il existe $k' \in \mathbb{N}$ et une configuration (ϵ, p', m') tels que $(c_1, p, m) \triangleright^{k'} (\epsilon, p', m')$ et $(c_2, p', m') \triangleright^{k-k'} (\epsilon, p_2, m_2)$.

La sémantique d'un programme cible

On définit la fonction sémantique

$$\mathcal{M} : \mathbf{Code} \rightarrow (\mathbf{State} \rightarrow \mathbf{State}).$$

$$\mathcal{M}[c]m = \begin{cases} m' & , (c, \epsilon, m) \triangleright^* (\epsilon, p, m') \\ \text{undef, sinon} \end{cases}$$

Génération de code : le problème

Nous voulons définir trois fonctions :

1. $\mathcal{CA} : \mathbf{Aexp} \rightarrow \mathbf{Code}$
2. $\mathcal{CB} : \mathbf{Bexp} \rightarrow \mathbf{Code}$
3. $\mathcal{CS} : \mathbf{Stm} \rightarrow \mathbf{Code}$

tels que pour tout programme $c \in \mathbf{Stm}$ on a

$$\mathcal{S}_{ns}[] = \mathcal{M} \circ \mathcal{CS}$$

Nous allons exiger que \mathcal{CA} , \mathcal{CB} et \mathcal{CS} satisfassent les propriétés suivantes :

1. $(\mathcal{CA}[a], \epsilon, \sigma) \triangleright^* (\epsilon, \mathcal{A}[a]\sigma, \sigma)$
2. $(\mathcal{CB}[b], \epsilon, \sigma) \triangleright^* (\epsilon, \mathcal{A}[b]\sigma, \sigma)$
3. $(\mathcal{CS}[S], \epsilon, \sigma) \triangleright^* (\epsilon, p, \sigma')$ ssi $(S, \sigma) \rightarrow \sigma'$

Génération de code

Quelques exemples de clauses pour définir \mathcal{CA} , \mathcal{CB} et \mathcal{CS} :

- $\mathcal{CA}[n] = \text{push-n}$.
- $\mathcal{CA}[x] = \text{fetch}(x)$
- $\mathcal{CA}[a_1 + a_2] = \mathcal{CA}[a_2] \cdot \mathcal{CA}[a_1] \cdot \text{add}$
- $\mathcal{CB}[\text{true}] = \text{True}$
- $\mathcal{CS}[x := a] = \mathcal{CA}[a] \cdot \text{store}(x)$
- $\mathcal{CS}[S_1; S_2] = \mathcal{CS}[S_1] \cdot \mathcal{CS}[S_2]$

Correction de la génération de code

Lemme

$$(\mathcal{CA}[a], \epsilon, \sigma) \triangleright^* (\epsilon, \mathcal{A}[a]\sigma, \sigma)$$

Par induction structurelle sur a

- **cas** $a = n$ (**push-n**, ϵ, σ) \triangleright^* ($\epsilon, \mathcal{N}[n], \sigma$)
- **cas** $a = x$ (**fetch**(x), ϵ, σ) \triangleright^* ($\epsilon, \sigma(x), \sigma$)
- **cas** $a = a_1 + a_2$
 $(\mathcal{CA}[a_2] : \mathcal{CA}[a_1] : \mathbf{add}, \epsilon, \sigma) \triangleright^* (\mathcal{CA}[a_1] : \mathbf{add}, \mathcal{A}[a_2]\sigma, \sigma) \triangleright^*$
 $(\mathbf{add}, \mathcal{A}[a_1]\sigma : \mathcal{A}[a_2]\sigma, \sigma) \triangleright^* (\epsilon, \mathcal{A}[a_1]\sigma + \mathcal{A}[a_2]\sigma, \sigma)$

Correction-2

Lemme : Si $(S, \sigma) \rightarrow \sigma'$ alors $(\mathcal{CS}[S], \epsilon, \sigma) \triangleright^* (\epsilon, \epsilon, \sigma')$

Par induction sur la forme de l'arbre de dérivation.

Lemme : Si $(\mathcal{CS}[S], \epsilon, \sigma) \triangleright^k (\epsilon, \epsilon, \sigma')$ alors $(S, \sigma) \rightarrow \sigma'$

Par induction sur k .