LSD^{12}: Et ça repart

Xavier Devroey Alain Solheid Michaël Marcozzi

Année académique 2011-2012

Table des matières

---

1 Introduction

1.1 La bête

Ce document présente le langage LSD^{12}, i.e. « Langage Simple et Didactique ». Le 12 signifie simplement 2012, l'année de mise en service du langage mais peut également signifier « douzième tentative ». Ce langage est donc l'héritier direct d'une grande famille composée de LSD^{02}, LSD^{03}, LSD^{04} , LSD^{05}, LSD^{06}, LSD^{07}, LSD^{08}, LSD^{09}, LSD^{10} et LSD^{11}.

Il est conçu pour être simple. Il n'est pas exceptionnel et ne révolutionnera pas l'informatique de demain. Mais il devrait vous permettre de comprendre correctement les différents concepts abordés au cours de Syntaxe et Sémantique. Ceci justifie l'aspect didactique du langage.

LSD^{12}vous permettra d'utiliser des booléens et des entiers comme types scalaires et des ensembles comme types composés. Le langage offre toutes les structures de contrôle présentes dans un bon langage impératif. Cela signifie, en particulier, que l'instruction « goto » a reçu la place qu'elle mérite : la poubelle. Le langage permet de définir des fonctions récursives et d'utiliser des passages de paramètres par valeur et par variable. Toute variable doit être déclarée avant son utilisation. La déclaration de la variable indique son type, qui ne peut être transformé lors de l'exécution.

1.2 Syntaxe, sémantique, domaine, à quoi ça sert ?

1.2.1 Syntaxe et grammaire

Comme expliqué au cours, la *syntaxe* du langage décrit l'ensemble des suites de symboles terminaux qui sont légales. Il est très important de connaître cette syntaxe, lorsque l'on utilise des langages de programmation. C'est ce qui vous permet d'écrire des programmes qui vont pouvoir être lus et traités par le compilateur. La syntaxe va donc décrire un ensemble Program. Par exemple, les phrases suivantes appartiennent toutes les deux à cet ensemble :

```
program x; function main():void; var a int; function fct0():void; var
   begin
write 3; end; begin fct0(); end; end;
```

et

```
program x;
  function main(): void;
   var
     { declaration de variables + fonctions }
     a  int;
     b  bool;

     function fct1(var a : int, b : bool) : void;
         forward;

     function fct0() : int;
         var
            a int;
         begin
            read a;
            return a;
         end;

     function fct1(var a : int, b : bool) : void;
         var
         begin
            if (b) then
               a := a+1;
            else
```

```
                a := a-1;
            fi;
        end;

    begin
        a := fct0();
        b := a <= 0;
        fct1(a,b);
        write a;
    end;
end;
```

Ces langages sont des ensembles très grands, typiquement infinis. Il faut donc être très (très !) patient si l'on veut énumérer leur contenu. Afin de décrire d'une manière finie (et donc, imprimable) le contenu de ces langages, nous utilisons des *grammaires non-contextuelles*. Pour rappel, une grammaire non-contextuelle est composée de quatre éléments :

1. V_T, un ensemble (fini) de symboles *terminaux*.
2. V_N, un ensemble (fini) de symboles *non-terminaux*.
3. $S \in V_N$, un non-terminal de départ.
4. $P \subseteq V_N \times (V_T \cup V_N)^*$, un ensemble de *productions*. Une production détermine par quelle suite de non-terminaux et de terminaux on a le droit de remplacer un non-terminal. Une production $(A, \alpha_1 \cdot \ldots \cdot \alpha_n)$ s'écrira plus lisiblement $A \to \alpha_1 \ldots \alpha_n$.

Par exemple, la grammaire des nombres en LSD^{12}serait :

1. $V_T = \{$ **0** , **1** , . . . , **9** , − $\}$, les caractères de 1 à 9 et le signe « - ».
2. $V_N = \{NbSig, Nb, Chiffre\}$
3. $S = NbSig$
4. les productions sont :

$$\begin{array}{rcl} NbSig & \to & Nb \\ NbSig & \to & - \; Nb \\ Nb & \to & Chiffre \\ Nb & \to & Nb \; Chiffre \\ Chiffre & \to & \mathbf{0} \\ Chiffre & \to & \mathbf{1} \\ & \vdots & \\ Chiffre & \to & \mathbf{9} \end{array}$$

Afin de déterminer quel langage est défini par une grammaire non-contextuelle, nous avons besoin du concept de *production*. D'abord, on dit qu'une grammaire *produit directement* un mot de symboles terminaux et/ou non-terminaux m', à partir d'un autre mot m, ssi on peut trouver :

1. un non-terminal A dans le mot m.
2. une production $A \to \phi$ dans la grammaire.

Tels que m' est identique à m sauf qu'une des occurrences de A a été remplacée par ϕ. Par exemple, la grammaire ci-dessus produit directement $-$ *NbChiffreNb* à partir du mot $-$ *NbNb*. Si la grammaire G produit directement m' à partir de m, on écrit : $m \Rightarrow_G m'$.

Une *dérivation* de m' à partir de m est une séquence finie de transformations comme ci-dessus. Pour dire que m' *se dérive de* m dans une grammaire G, il faut donc trouver une séquence de mots $m_1, \ldots, m_n$ telle que :

1. $m = m_1$
2. $\forall i : 1 < i \leq n : m_{i-1} \Rightarrow_G m_i$
3. $m_n = m'$

Cette séquence de mots est appelée une *dérivation*. Si il existe une dérivation de m' à partir de m selon G, nous écrirons $m \Rightarrow_G^* m'$

Étant donné une grammaire G, son langage $\mathcal{L}_G$, qui est un ensemble de suites de symboles terminaux, $\mathcal{L}_G \subseteq V_T^*$, est défini comme l'ensemble des phrases de terminaux qui sont dérivables du symbole initial de G.

$$\mathcal{L}_G = \{m \in V_T^* | S \Rightarrow_G^* m\}$$

L'occurrence d'un non-terminal qui est remplacé est choisie *au hasard*[1]. Si l'on s'oblige à toujours transformer la *première* occurrence d'un non-terminal dans m, il s'agit d'une dérivation gauche et, si l'on doit toujours d'abord traiter le *dernier* non-terminal de m, il s'agit d'une dérivation droite.

Par exemple, le mot $-$ **1** **3** appartient au langage des nombres défini ci-dessus. Par contre, les mots $-$ $-$ **2** et $-$ *Chiffre* n'y appartiennent pas. Le premier, parce-qu'il n'y a pas de dérivation de *NbSig* qui permette d'obtenir ce mot, et le second, parce-qu'il contient un non-terminal.

En résumé, il faut retenir que la syntaxe d'un langage de programmation est un ensemble infini de suites d'éléments terminaux qui est décrit de manière finie par une grammaire non-contextuelle.

Program est l'ensemble des programmes LSD^{12}syntaxiquement valides. La grammaire décrivant Program est donnée tout au long de ce document.

1.2.2 Sémantique

Les éléments appartenant à Program n'ont pas de signification. Le but de la *définition sémantique* est de leur fournir cette signification. Pour cela, on procède en deux étapes. D'abord, on fournit un *domaine sémantique*, qui est l'ensemble des significations possibles. Ensuite, on donne une fonction qui fait correspondre, à chaque élément du langage, un élément du domaine sémantique.

Un exemple devrait éclairer cette obscure explication : prenons le langage des nombres LSD^{12}, défini dans la section ci-dessus. L'ensemble des nombres syntaxiquement valides est appelé Nb. Il nous faut d'abord trouver un domaine sémantique, c'est-à-dire un ensemble de valeurs que *représentent* les éléments de Nb. Nous choisissons $\mathbb{Z}$, l'ensemble des nombres entiers, positifs, négatifs ou nuls. Ensuite, nous devons expliquer comment interpréter les éléments syntaxiques. Nous devons donc faire correspondre, à chaque élément de Nb, un élément de $\mathbb{Z}$. Vu que Nb et $\mathbb{Z}$ sont infinis, nous ne pouvons pas, évidemment, écrire une longue table de correspondance, comme la table 1.

Précédemment, nous avons expliqué que l'on pouvait décrire les langages, qui sont des ensembles infinis, de manière finie, en utilisant des grammaires. Partant du même principe, nous allons définir, de manière finie, les fonctions qui, à chaque élément du domaine syntaxique, font correspondre un élément du domaine sémantique. Pour obtenir cette définition finie, nous utiliserons typiquement une induction sur la structure des éléments syntaxiques.

1. Techniquement, on dit « non-déterministiquement », càd que le hasard fait toujours bien les choses ; la dérivation correcte, si elle existe, est toujours choisie

TABLE 1 – Une très longue table de correspondance

Nb	$\mathbb{Z}$
0	0
1	1
− 1	−1
⋮	⋮
1578	1578
⋮	⋮

La manière de procéder sera donc la suivante. Pour chacune des formes possibles des termes du domaine syntaxique, nous dirons comment l'élément du domaine sémantique peut être calculé. Le grand principe qui soutient la *sémantique dénotationnelle* est la *compositionnalité* : « la sémantique d'un élément ne peut être définie qu'en fonction de la sémantique de ses composants directs ».

Par exemple, si l'on se réfère à Nb, l'ensemble des nombres LSD^{12}, nous avons décidé que le domaine sémantique était $\mathbb{Z}$. La fonction qui fera correspondre à chaque élément de Nb un élément de $\mathbb{Z}$ s'appelle : $\mathcal{N}$. Elle possède la signature suivante :

$$\mathcal{N} : \mathsf{Nb} \to \mathbb{Z}$$

La flèche $\to$ signifie que $\mathcal{N}$ est une fonction totale : elle est définie pour tous les éléments de Nb.

Nous définissons $\mathcal{N}$ par les équations ci-dessous. Notez l'utilisation des doubles crochets $[\![$ et $]\!]$. Ils ne servent qu'à encadrer les éléments qui appartiennent au domaine syntaxique, de façon à mettre en évidence la distinction entre éléments *syntaxiques* et *sémantiques*.

$$\begin{array}{rcl} \mathcal{N}[\![\textbf{ − } n]\!] & = & 0 - \mathcal{N}[\![n]\!] \\ \mathcal{N}[\![n \cdot c]\!] & = & (10 * \mathcal{N}[\![n]\!]) + \mathcal{N}[\![c]\!], \text{ avec } n \in \mathit{Chiffre}^+ \\ \mathcal{N}[\![\textbf{ 0 }]\!] & = & 0 \\ \mathcal{N}[\![\textbf{ 1 }]\!] & = & 1 \\ & \vdots & \\ \mathcal{N}[\![\textbf{ 9 }]\!] & = & 9 \end{array} \tag{1}$$

Si il s'agit d'une suite de chiffres n, précédés d'un signe **−** , alors, l'entier correspondant est l'opposé de l'entier représenté par n. Si n est un simple chiffre, alors, on donne, par une liste *finie*, l'entier qui lui correspond. Par exemple, **3** correspondra à 3. Si n peut être divisé en deux parties : n' et c telles que n' est une suite de chiffres (non vide) et c est un chiffre, alors, il faut "décaler vers la gauche" l'entier représenté par n' et y ajouter la valeur de c dans la partie des unités.

1.2.3 Oui, mais . . . à quoi ça sert ?

L'utilité de la description syntaxique du langage est indiscutable. Elle permet au programmeur ou au constructeur d'un compilateur, de déterminer, sans discussion possible, quelles constructions sont légales dans le langage et lesquelles ne le sont pas.

L'utilité de la définition sémantique peut sembler moins frappante. Pourtant, il suffit de réfléchir aux nombreux choix que vous pouvez poser si vous devez interpréter un appel de procédure. Comment sont passés les paramètres ? Que signifie « passage par adresse » ou « passage par résultat » ? Dans

quel ordre dois-je évaluer les composantes d'une expression ? Avoir une sémantique formelle permet de répondre à ces questions, car elle n'induit qu'une seule interprétation possible des programmes écrits dans un langage.

Donc, dans un monde idéal, les langages seraient tous dotés d'une sémantique formelle, que les programmeurs consulteraient avant de construire les compilateurs. De cette manière, un programme d'un langage $\mathcal{L}$ donnerait les mêmes résultats, qu'il soit compilé et exécuté sur une plate-forme X ou Y.

Votre travail, lors de ce TP, sera de construire un *compilateur*. Un compilateur est un programme qui, prenant en entrée un programme P_s, écrit dans le langage *source* Source, produit en sortie un programme P_o dans le langage *objet*, Obj. Cette traduction doit *préserver la sémantique de* P_s. Formellement, étant donné un domaine sémantique $\mathbb{D}$ et deux fonctions sémantiques : $\mathcal{S}_s : \mathsf{Source} \to \mathbb{D}$ et $\mathcal{S}_o : \mathsf{Obj} \to \mathbb{D}$, si P_o est le programme compilé à partir de P_s, alors,

$$\mathcal{S}_s[\![P_s]\!] = \mathcal{S}_o[\![P_o]\!]$$

Le compilateur que vous devez construire pourra traduire n'importe quel programme LSD^{12} syntaxiquement valide et respectant les contraintes sur les identificateurs en un programme `P-Code` équivalent. De plus, votre compilateur devra refuser de traduire des textes qui ne sont pas des programmes LSD^{12} syntaxiquement valides ou qui violent les contraintes sur les identificateurs.

Les contraintes sur les identificateurs ne sont pas considérées exactement comme des contraintes syntaxiques, car elles ne peuvent être exprimées au moyen d'une grammaire non-contextuelle.

1.3 Conventions de notation

Au point de vue syntaxique, les terminaux seront notés en police **`courrier gras`** , comme dans e_1 **+** e_2. Il ne faudra donc pas confondre **+** et +. Le premier représente le terminal « + » qui est un caractère tandis que le deuxième est le signe représentant l'opération d'addition sur les entiers, avec sa sémantique usuelle.

Les ensembles syntaxiques sont écrits en caractère sans sérif, comme Nb. Les fonctions sémantiques en caractère « calligraphié », comme $\mathcal{I}$, et les domaines sémantiques avec des majuscules à double barre, comme dans $\mathbb{Z}$.

Lorsque nous écrivons des grammaires, nous utilisons une forme abrégée des grammaires non-contextuelles, où $(,)[,], ^+, ^*, |$ sont des méta-symboles (des « caractères réservés » de la grammaire non-contextuelle).

- $\phi_1|\phi_2$ représente le choix entre ϕ_1 et ϕ_2
- $[\phi_1]$ représente le fait que ϕ_1 soit optionnel
- ϕ_1^+ correspond à n répétitions de ϕ_1 (avec $n \geq 1$)
- ϕ_1^* représente n répétitions de ϕ_1 (avec $n \geq 0$)

1.4 Structure du document

Dans ce document, la syntaxe et la sémantique du langage vont vous être présentées. Plutôt que de diviser la présentation en donnant d'abord toute la syntaxe et ensuite toute la sémantique, nous travaillons par « petits bouts », en remontant des éléments les plus simples du langage (les expressions) vers l'élément le plus important (le programme).

2 Domaine sémantique

Un programme LSD^{12}est un transformateur de suites d'entiers. Il reçoit en entrée une suite finie d'entiers et produit en sortie une suite d'entiers. Il se peut également qu'il produise une erreur.

Afin de transformer son entrée en la sortie désirée, le programme manipule deux structures de données : une mémoire et un environnement. La mémoire assigne des valeurs à des adresses. Certaines adresses ne sont pas utilisées. La mémoire sera donc modélisée par une fonction *partielle*. Une fonction totale de A dans B est représentée par $A \to B$ tandis qu'une fonction partielle est écrite $A \hookrightarrow B$. Afin de simplifier la présentation et d'abstraire les choix d'implémentation, nous supposerons que la mémoire est infinie. Une valeur peut être soit une valeur scalaire, soit un ensemble.

L'environnement permet d'obtenir l'adresse correspondant à chaque identificateur de variable apparaissant dans le programme. Il permet aussi de retrouver la fonction associée à chaque identificateur de fonction. Il est important de se rendre compte qu'un identificateur peut être attaché à différentes adresses, en fonction du contexte dans lequel il est évalué. Un environnement sera donc une suite de fonctions, appelées contextes : $\mathsf{Id} \hookrightarrow \mathbb{A} \cup \mathbb{P}$. Habituellement, lorsque l'on décrit des suites (ou des mots), on utilise la notation "$\cdot$" pour représenter la concaténation de deux séquences ; nous ne dérogerons pas à cette tradition. Soit un environnement $e = f_1 \cdot \ldots \cdot f_n$. e peut être vu comme une pile, où les contextes les plus récents sont "au-dessus" (position n) et les contextes les plus vieux "en-dessous" (position 1).

$$e = \begin{pmatrix} f_n \\ f_{n-1} \\ \vdots \\ f_2 \\ f_1 \end{pmatrix}$$

Pour trouver l'élément auquel un identificateur id correspond dans l'environnement e, on regarde d'abord s'il est défini dans le contexte sur le dessus de la pile. Si id n'est pas défini dans le contexte courant (celui au-dessus de la pile), on regarde s'il est défini dans un contexte antérieur.

$$\begin{array}{lcll} e(id) & = & f_n(id) & \text{si} \quad f_n(id) \neq \bot \vee n = 1 \\ e(id) & = & (f_1 \cdot \ldots \cdot f_{n-1})(id) & \text{sinon} \end{array}$$

Par exemple, en supposant que l'on ait un environnement e défini comme ci-dessous et que l'ensemble des identificateurs soit $\{x, y\}$

$$e = [x \mapsto 123, y \mapsto \bot] \cdot [x \mapsto g, y \mapsto 256] \cdot [x \mapsto \bot, y \mapsto \text{ vrai }]$$

Alors $e(x) = g$ et $e(y) =$ vrai .

Formellement, nous utiliserons donc les ensembles suivants :

$\mathbb{Z}$ représente l'ensemble des entiers.

$\mathbb{B}$ représente l'ensemble des booléens : $\mathbb{B} = \{$ vrai , faux $\}$.

$\mathbb{I}Sets$ est l'ensemble des ensembles d'entiers : $\mathbb{I}Sets = 2^{\mathbb{Z}}$.

$\mathbb{A}$ est l'ensemble des adresses. Nous posons $\mathbb{A} = [1..max]$ où max est supposé suffisamment grand pour permettre l'exécution de n'importe quel programme LSD^{12}.

$\mathbb{V}$ est l'ensemble des valeurs ($\mathbb{V} = \mathbb{B} \cup \mathbb{Z} \cup \mathbb{I}Sets$),

$\mathbb{E}$ est l'ensemble des environnements. Dans LSD^{12}les choses sont un peu particulières car dans un environnement donné, on peut trouver une variable et une fonction partageant le même identifiant. Les environnement sont en fait des piles de la forme : $(\mathsf{Id} \times \mathbb{C}ategorie \hookrightarrow \mathbb{F} \cup \mathbb{A} \cup \mathbb{T}ypes)^{+}$ [2] , où $\mathbb{C}ategorie = \{$ vp , func $\}$, $\mathbb{T}ypes = \{$ Int , Bool , IntSet $\}$ et Id est l'ensemble des identifiants défini dans la section suivante.

$\mathbb{S}$ est l'ensemble des mémoires, càd des fonctions de signature $\mathbb{A} \hookrightarrow \mathbb{V} \cup \{$ Uninit $\}$. Certaines adresses sont allouées mais non initialisées.

$\mathbb{I}$ **et** $\mathbb{O}$ sont les ensembles d'inputs et d'outputs, par définition, $\mathbb{I} = \mathbb{O} = \mathbb{Z}^{*}$,

$\mathbb{S}tate$ est l'ensemble des *états* du programme. Un état doit contenir toute l'information qui permet de calculer l'effet d'une instruction. Nous définissons $\mathbb{S}tate$ comme $\mathbb{S} \times \mathbb{E} \times \mathbb{I} \times \mathbb{O}$.

$\mathbb{R}es$ est l'ensemble des *résultats* de l'évaluation de fonctions. Lorsqu'une fonction est évaluée, elle modifie l'état courant, par effets de bords, et retourne une valeur. $\mathbb{R}es = \mathbb{V} \times \mathbb{S}tate$

$\mathbb{F}$ est l'ensemble qui représente les fonctions sémantiques. Chaque fonction LSD^{12} correspondra à un élément de $\mathbb{F}$. Lorsque l'on invoque une fonction, l'état est modifié et une valeur peut être retournée. Le résultat de l'appel dépend de deux choses : l'état dans lequel la fonction est appelée et la valeur des paramètres effectifs. Il est possible que la fonction se plante, ce qui explique pourquoi $\mathbb{F}$ est partielle sur $\mathbb{R}es$.

$$\mathbb{F} : \mathbb{S}tate \rightarrow \mathbb{V}^{*} \hookrightarrow \mathbb{R}es \cup \mathbb{S}tate$$

3 Identificateurs

3.1 Syntaxe

Un identificateur (*Id*) est un élément appartenant à $\mathsf{Id} = \mathsf{Mot} \setminus \mathsf{Keywords}$ où la catégorie syntaxique Mot est définie par la grammaire ci-dessous et la catégorie syntaxique Keywords contient tous les mots-cles du langage LSD^{12}. Nous ne définissons pas ces mots-clefs en extension, mais le lecteur les découvrira en parcourant les différentes règles.

$$\begin{aligned} \textit{Mot} &\rightarrow \textit{Lettre} \ (\ \textit{Chiffre} \mid \textit{Lettre}\ )^{*} && (2)\\ \textit{Lettre} &\rightarrow \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \texttt{d} \mid \texttt{e} \mid \texttt{f} \mid \texttt{g} \mid \texttt{h} \mid \texttt{i} \mid \texttt{j} \mid \texttt{k} \mid \texttt{l} \mid \texttt{m} \\ &\quad \mid \texttt{n} \mid \texttt{o} \mid \texttt{p} \mid \texttt{q} \mid \texttt{r} \mid \texttt{s} \mid \texttt{t} \mid \texttt{u} \mid \texttt{v} \mid \texttt{w} \mid \texttt{x} \mid \texttt{y} \mid \texttt{z} \\ &\quad \mid \texttt{A} \mid \texttt{B} \mid \texttt{C} \mid \texttt{D} \mid \texttt{E} \mid \texttt{F} \mid \texttt{G} \mid \texttt{H} \mid \texttt{I} \mid \texttt{J} \mid \texttt{K} \mid \texttt{L} \mid \texttt{M} \\ &\quad \mid \texttt{N} \mid \texttt{O} \mid \texttt{P} \mid \texttt{Q} \mid \texttt{R} \mid \texttt{S} \mid \texttt{T} \mid \texttt{U} \mid \texttt{V} \mid \texttt{W} \mid \texttt{X} \mid \texttt{Y} \mid \texttt{Z} && (3) \end{aligned}$$

3.2 Sémantique

Le domaine sémantique des identificateurs est un petit peu particulier. Il s'agit de l'ensemble (syntaxique) des identificateurs. Un identificateur sera interprété par ... lui-même ! La fonction sémantique sera simplement *id*, la fonction identité ($\forall x : id(x) = x$).

2. en fait, c'est une pile d'éléments de l'union $(\mathsf{Id} \times \{$ vp $\} \rightarrow \mathbb{A}) \cup (\mathsf{Id} \times \{$ func $\} \rightarrow \mathbb{F}) \cup (\mathsf{Id} \rightarrow \mathbb{T}ypes)$

4 Expressions droites

Les expressions droites sont des expressions dont l'évaluation retourne une valeur (i.e. un élément de $\mathbb{V}$). Elles portent ce nom car elles apparaissent dans la partie *droite* des instructions d'affectation. Une expression droite peut modifier, par effet de bords, l'état courant.

L'évaluation d'une expression peut échouer, parce que l'on se réfère à une variable entière qui n'existe pas, par exemple. Si la valeur d'une variable est Uninit , nous supposerons que son utilisation conduira à une erreur. La sémantique d'une expression droite sera donc donnée par une fonction *partielle*. Cette fonction sera donc définie si l'évaluation réussit. Il est particulièrement important de faire attention au *type* de ces expressions.

4.1 Syntaxe

Une expression droite est soit une expression de type adresse, entière ou booléenne. On retrouve ci-dessous les expressions classiques : addition, soustraction, multiplication, division entière, opérations booléennes et accès à la valeur stockée à une adresse (*ExprG*). En sus, on retrouve des opération propre aux ensembles d'entiers : l'union, l'intersection, la différence, l'extraction de la valeur minimum de l'ensemble d'entiers, l'extraction de la valeur maximum de l'ensemble d'entiers, le nombre d'éléments que compte un ensemble d'entiers et le test de la présence d'une valeur dans cet ensemble d'entiers.

La grammaire suivante définit une expression droite (*ExprD*) comme étant une expression entière (*ExprEnt*) ou une expression booléenne (*ExprBool*) :

$$\begin{array}{lll} \textit{ExprD} & \rightarrow & \textit{ExprEnt} \\ & & \mid \textit{ExprBool} \end{array}$$

Une expression entière (*ExprEnt*) sera définie comme un nombre, une opération sur des entiers (**+** **−** ***** **/**), une opération sur un ensemble d'entiers retournant un entier (**min** **max** **#**), une expression gauche (*ExprG* , i.e. une variable), un appel de fonction [3] (*Id* **(** [*ExprD* (**,** *ExprD*)*] **)**) ou une expression entière placée entre parenthèses (**(** *ExprEnt* **)**) :

$$\begin{array}{lll} \textit{ExprEnt} & \rightarrow & \textit{NbSig} \\ & & \mid \textit{ExprEnt}\ \textbf{+}\ \textit{ExprEnt} \\ & & \mid \textit{ExprEnt}\ \textbf{−}\ \textit{ExprEnt} \\ & & \mid \textit{ExprEnt}\ \textbf{*}\ \textit{ExprEnt} \\ & & \mid \textit{ExprEnt}\ \textbf{/}\ \textit{ExprEnt} \\ & & \mid \textbf{min}\ \textit{ExprG} \\ & & \mid \textbf{max}\ \textit{ExprG} \\ & & \mid \#\ \textit{ExprG} \\ & & \mid \textit{ExprG} \\ & & \mid \textit{Id}\ \textbf{(}\ [\ \textit{ExprD}\ (\ \textbf{,}\ \textit{ExprD}\ )^*]\ \textbf{)} \\ & & \mid \textbf{(}\ \textit{ExprEnt}\ \textbf{)} \end{array}$$

3. Lors de la vérification des types, le compilateur devra s'assurer que la fonction retourne bien une valeur entière.

Une expression booléenne (*ExprBool*) sera définie comme une constant (**true** ou **false**), une opération sur des booléens (**&&** **||** **!**), la comparaison de deux entiers (**=** **<** **<=**), le test de la présence d'un entier dans un ensemble d'entiers (**in**) , une expression gauche (*ExprG* , i.e. une variable), un appel de fonction[4] (*Id* **(** [*ExprD* (**,** *ExprD*)*] **)**) ou une expression booléenne placée entre parenthèses (**(** *ExprBool* **)**) :

$$
\begin{aligned}
\mathit{ExprBool} \quad \rightarrow \quad & \texttt{false} \\
& \mid \texttt{true} \\
& \mid \mathit{ExprBool}\ \texttt{\&\&}\ \mathit{ExprBool} \\
& \mid \mathit{ExprBool}\ \texttt{||}\ \mathit{ExprBool} \\
& \mid \texttt{!}\ \mathit{ExprBool} \\
& \mid \mathit{ExprEnt}\ \texttt{=}\ \mathit{ExprEnt} \\
& \mid \mathit{ExprEnt}\ \texttt{<}\ \mathit{ExprEnt} \\
& \mid \mathit{ExprEnt}\ \texttt{<=}\ \mathit{ExprEnt} \\
& \mid \mathit{ExprEnt}\ \texttt{in}\ \mathit{ExprG} \\
& \mid \mathit{ExprG} \\
& \mid \mathit{Id}\ \texttt{(}\ [\ \mathit{ExprD}\ (\ \texttt{,}\ \mathit{ExprD}\ )^*]\ \texttt{)} \\
& \mid \texttt{(}\ \mathit{ExprBool}\ \texttt{)}
\end{aligned}
$$

4.1.1 Priorité des opérateurs

Afin de mettre un programme LSD^{12}sous une forme non-ambigüe, il faut connaître les règles de précédence des différents opérateurs. Nous choisissons la règle suivante :

$$\{\ \texttt{\&\&}\ ,\ \texttt{||}\ \} < \{\ \texttt{!}\ \} < \{\ =\ ,\ \texttt{<=}\ ,\ \texttt{<}\ ,\ \texttt{in}\ \} < \{\ \texttt{+}\ ,\ -\ \} < \{\ \texttt{*}\ ,\ \texttt{/}\ \} < \{\ \texttt{min}\ ,\ \texttt{max}\ ,\ \#\ \} < \{\ \texttt{(}\ .\ \texttt{)}\ \}$$

Où $b < a$ signifie a doit être évalué avant b. Toutes les opérations sont associatives à gauche : on lit les termes de gauche à droite. Par exemple, l'expression : $\texttt{x} + \texttt{3} * \texttt{y} - \texttt{5} - \texttt{z}$ doit être interprétée comme $\Big(\big(\texttt{x} + (\texttt{3} * \texttt{y})\big) - \texttt{5}\Big) - \texttt{z}$.

Un autre exemple : **true||false&&true** sera interprété comme : **(true||false)&&true** . Un troisième exemple plus complexe :

$$\texttt{!}\ x = \texttt{3} * \texttt{5}\ \ \texttt{\&}\ \ \texttt{true}\ \ \texttt{||}\ y \texttt{<=}\ x$$

Sera interprété comme :

$$\Big(\big(\ \texttt{!}\ (x = (\ \texttt{3} * \texttt{5}\ ))\big)\ \texttt{\&\&}\ \ \texttt{true}\ \Big)\ \texttt{||}\ (y \texttt{<=}\ x)$$

4. Lors de la vérification des types, le compilateur devra s'assurer que la fonction retourne bien une valeur booléenne. **A noter ici qu'une fonction ne pourra retourner qu'un entier ou un booléen, pas un ensemble d'entier !**

4.1.2 Résolution des conflits

Si elles sont passées tel quel à un générateur d'analyseur syntaxique comme bison (YACC), les règles *ExprEnt* et *ExprBool* vont générer des conflits reduce/reduce. Pratiquement, cela signifie que lorsque l'analyseur va rencontrer une certaine séquence de terminaux, il ne saura pas quel règle appliquer.

Par exemple, si on considère la séquence suivante `fct(1)`, l'analyseur syntaxique est incapable de déterminer si il s'agit d'une *ExprEnt* ou d'une *ExprBool* . Il en va de même pour les *ExprG* : lorsqu'il rencontrera un terminal `a`, correspondant à une variable, l'analyseur ne pourra déterminer si il s'agit d'une *ExprEnt* ou d'une *ExprBool* .

Une manière de lever ce conflit est de ramener les parties droites des règles définissant une *ExprEnt* et une *ExprBool* dans la parie droite de la règle définissant une *ExprD* et de laisser la vérification des types des opérandes des différents opérateurs au compilateur. Une expression droite peut dont être redéfinie comme :

$$
\begin{array}{lcl}
\mathit{ExprD} & \rightarrow & \mathit{ExprG} \\
 & & \mid \mathit{Id}\ \ \mathbf{(}\ [\ \mathit{ExprD}\ (\ \mathbf{,}\ \ \mathit{ExprD}\ )^{*}]\ \mathbf{)} \\
 & & \mid \mathbf{true} \\
 & & \mid \mathbf{false} \\
 & & \mid \mathbf{NbSig} \\
 & & \mid \mathit{ExprD}\ \ \mathbf{+}\ \ \mathit{ExprD} \\
 & & \mid \ldots
\end{array}
$$

4.2 Vérification des types

Avant d'évaluer une expression, nous demandons que son type soit vérifié. Une erreur de type est une erreur de sémantique statique qui doit être détectée lors de l'analyse syntaxique. Un programme comprenant une erreur de type sera rejeté *avant exécution* par un compilateur LSD^{12}.

Dans la suite, nous décrirons également comment calculer le type d'une expression gauche. Pour l'instant, nous faisons l'hypothèse que nous l'obtenons grâce à la fonction $\mathcal{T}ype_G$. De quelle information avons-nous besoin pour calculer le type d'une expression gauche ?

1. Le type de chaque identifiant de variable ou de fonction (et dans la suite de paramètre). Nous utiliserons pour ce faire une fonction $ty : \mathsf{Id} \times \mathbb{C}ategorie \rightarrow \mathbb{T}ypes$. Pour un identifiant x et sa catégorie c, ty renvoie son type, nous appellerons parfois cette fonction ty un dictionnaire.
2. Le type des paramètres (la "signature") de chaque fonction utilisée. En effet, dans un cas de figure comme **`f(y) in x`** , il faudra vérifier que y est du bon type par rapport à la déclaration de f, que $f(y)$ renvoie bien un entier et que x est bien un ensemble d'entiers. Nous utiliserons une fonction $sig : \mathsf{Id} \rightarrow \mathbb{T}ypes^{*}$.

Dès lors, nous pouvons décrire la fonction

$$\mathcal{T}ype : ((\mathsf{Id} \times \mathbb{C}ategorie) \rightarrow \mathbb{T}ypes) \rightarrow (\mathsf{Id} \hookrightarrow \mathbb{T}ypes^{*}) \rightarrow \mathsf{ExprD} \hookrightarrow \mathbb{T}ypes$$

Dans la définition ci-dessous, nous utilisons la notation *curryfiée*. L'expression $\mathcal{T}ype\ ty\ sig[\![x]\!]$ signifie simplement que, étant donné ty et sig, la fonction $\mathcal{T}ype$ calcule le type de x.

Par exemple, si nous utilisons ces fonctions pour définir le typage d'une expression droite placée entre parenthèses ((x)), cela donne :

$$\mathcal{T}ype\ ty\ sig\ [\![\ (\ x\ )\ ]\!] = \mathcal{T}ype\ ty\ sig\ [\![x]\!]$$

Signifiant que le type de l'expression (x) équivaudra au type de x.

4.2.1 Typage des expressions gauches et des appels de fonctions

$$\mathcal{T}ype\ ty\ sig\ [\![x]\!]\ = \mathcal{T}ype_G\ ty\ sig\ [\![x]\!] \quad \text{où} \quad x \in \mathsf{ExprG}$$

$$\mathcal{T}ype\ ty\ sig\ [\![x\ (\ el\ )\ ]\!] = ty(x,\ \text{func}\ ) \quad \text{si} \quad map\ (\mathcal{T}ype\ ty\ sig, el) = sig(x)$$

La fonction *map* utilisée ci-dessus est bien connue des étudiants ayant programmé en Haskell : elle applique une fonction à tous les éléments d'une liste. Par exemple, en supposant que $succ(x) = x+1$, $map(succ, 3 \cdot 4 \cdot 2) = 4 \cdot 5 \cdot 3$. Ici, nous calculons le type de chaque argument lors de l'appel d'une fonction et vérifions que la liste résultante est la signature de la fonction.

4.2.2 Typage des expressions entières

La vérification du typage pour une constante entière est triviale :

$$\text{pour } x \in \mathsf{NbSig}, \mathcal{T}ype\ ty\ sig\ [\![x]\!] = \text{ Int}$$

Pour les opérations d'addition, de soustraction, de division et de multiplication, l'expression est bien typée si chacune des deux opérandes est de type entier :

$$\text{pour } op \in \{\ \texttt{*}\ ,\ \texttt{/}\ ,\ \texttt{+}\ ,\ \texttt{-}\ \},$$
$$\mathcal{T}ype\ ty\ sig\ [\![x\ op\ y]\!] = \text{ Int} \quad \text{si} \quad \mathcal{T}ype\ ty\ sig\ [\![x]\!] = \text{ Int } = \mathcal{T}ype\ ty\ sig\ [\![y]\!]$$

Les opérations minimum (**min**), maximum (**max**) et cardinal (#) d'un ensemble d'entiers sont bien typées pour autant que leurs opérandes soient de type ensemble d'entiers :

$$\text{pour } op \in \{\ \textbf{min}\ ,\ \textbf{max}\ ,\ \#\ \},$$
$$\mathcal{T}ype\ ty\ sig\ [\![op\ x]\!] = \text{ Int} \quad \text{si} \quad \mathcal{T}ype\ ty\ sig\ [\![x]\!] = \text{ IntSet}$$

4.2.3 Typage des expressions booléennes

La vérification de type sur les constantes **true** et **false** est triviale :

$$\text{pour } x \in \{\ \textbf{true}\ ,\ \textbf{false}\ \},\ \mathcal{T}ype\ ty\ sig\ [\![x]\!] = \text{ Bool}$$

Les opérations logiques (**&&** , **||** et **!**) sont bien typées pour autant que leurs opérandes soient des booléens :

$$\text{pour } op \in \{\ \texttt{\&\&}\ ,\ \texttt{||}\ \}$$
$$\mathcal{T}ype\ ty\ sig\ [\![x\ op\ y]\!] = \text{ Bool} \quad \text{si} \quad \mathcal{T}ype\ ty\ sig\ [\![x]\!] = \text{ Bool } = \mathcal{T}ype\ ty\ sig\ [\![y]\!]$$

$$\mathcal{T}ype\ ty\ sig\ [\![\ \texttt{!}\ x]\!] = \text{ Bool} \quad \text{si} \quad \mathcal{T}ype\ ty\ sig\ [\![x]\!] = \text{ Bool}$$

Les opérateurs de comparaison (**=** , **<=** , **<**) sont bien typés pour autant qu'ils portent sur des entiers :

pour $op \in \{$ `=` , `<=` , `<` $\}$,

$\mathcal{T}ype\ ty\ sig\ [\![x\ op\ y]\!] =$ Bool si $\mathcal{T}ype\ ty\ sig\ [\![x]\!] =$ Int $= \mathcal{T}ype\ ty\ sig\ [\![y]\!]$

L'opération de test de la présence d'une valeur dans un ensemble d'entiers (**`in`**) est bien typée si son opérande gauche est de type entier et si son opérande droite est de type ensemble d'entiers :

$$\begin{array}{rlll} \mathcal{T}ype\ ty\ sig\ [\![x\ \textbf{in}\ y]\!] = & \text{Bool} & \text{si} & \mathcal{T}ype\ ty\ sig\ [\![x]\!] = \text{Int} \\ & & & \mathcal{T}ype\ ty\ sig\ [\![y]\!] = \text{IntSet} \end{array}$$

4.3 Sémantique

La sémantique d'une expression droite est donnée par la fonction :

$$\mathcal{E} : \mathsf{ExprD} \rightarrow\ \mathbb{S}tate\ \hookrightarrow\ \mathbb{R}es$$

Pour rappel, $\mathbb{R}es$ est défini comme $\mathbb{R}es\ =\ \mathbb{V}\ \times\ \mathbb{S}tate$, ce qui signifie donc que les résultats de l'application de la fonction sémantique $\mathcal{E}$ seront des couples (v, s) où v représente une valeur et s un état du programme.

On évalue les opérandes de gauche à droite, ce qui peut modifier (par effets de bord) l'état du programme. C'est pourquoi la deuxième opérande sera évaluée dans l'état résultant de l'évaluation de la première opérande. Enfin, la sémantique n'est définie que si la sémantique des composants est définie. Ce qui signifie que en cas de "plantage", l'évaluation d'une expression droite est donc indéfinie.

Comme pour la vérification des types, la sémantique se définit de manière récursive. Par exemple, la sémantique d'une expression droite entre parenthèses (x) correspondra à la sémantique de l'expression droite x :

$$\mathcal{E}[\![\ (\ x\ )\ ]\!]s\ \ =\ \ \mathcal{E}[\![x]\!]$$

4.3.1 Sémantique des expressions entières

Nous définissons d'abord la sémantique des constantes. Ceci est immédiat : il s'agit seulement de renvoyer leur homologue sémantique, sans modifier l'état courant.

$$\mathcal{E}[\![x]\!]s = (\mathcal{N}[\![x]\!], s) \text{ où } x \in \mathsf{NbSig}$$

Pour les opérations binaires sur des expressions entières, la sémantique sera :

$$\begin{array}{rcll} \mathcal{E}[\![x\ \texttt{+}\ y]\!]s & = & (v_x + v_y, s_2) & \text{où} \quad \mathcal{E}[\![x]\!]s = (v_x, s_1) \\ & & & \quad\ \ \mathcal{E}[\![y]\!]s_1 = (v_y, s_2) \\ \mathcal{E}[\![x\ \texttt{-}\ y]\!]s & = & (v_x - v_y, s_2) & \text{où} \quad \mathcal{E}[\![x]\!]s = (v_x, s_1) \\ & & & \quad\ \ \mathcal{E}[\![y]\!]s_1 = (v_y, s_2) \\ \mathcal{E}[\![x\ \texttt{*}\ y]\!]s & = & (v_x * v_y, s_2) & \text{où} \quad \mathcal{E}[\![x]\!]s = (v_x, s_1) \\ & & & \quad\ \ \mathcal{E}[\![y]\!]s_1 = (v_y, s_2) \\ \mathcal{E}[\![x\ \texttt{/}\ y]\!]s & = & (v_x\ /\ v_y, s_2) & \text{où} \quad \mathcal{E}[\![x]\!]s = (v_x, s_1) \\ & & & \quad\ \ \mathcal{E}[\![y]\!]s_1 = (v_y, s_2) \text{ et } v_y \neq 0 \end{array}$$

La sémantique de `+` , `-` , `*` et `/` correspond donc à :

1. L'évaluation de l'opérande gauche dans l'état s, qui va amener le programme de l'état s à l'état s_1.
2. L'évaluation de l'opérande droite dans l'état s_1, qui va amener le programme de l'état s_1 à l'état s_2.
3. L'application de l'opération (`+` , `-` , `*` ou `/`) dans l'état s_2 sur les résultats des évaluations des deux opérandes.

Pour ce qui est des opérations **`min`** et **`max`** portant sur des ensembles et retournant des entiers, la sémantique est :

$$\mathcal{E}[\![\textbf{min}\ x]\!]s = (v_m\ tq((\ v_m \in v_x) \wedge (v_m \leq v_i \forall v_i \in v_x)), s_1) \quad \text{où} \quad \mathcal{E}[\![x]\!]s = (v_x, s_1)$$

$$\mathcal{E}[\![\textbf{max}\ x]\!]s = (v_m\ tq((\ v_m \in v_x) \wedge (v_m \geq v_i \forall v_i \in v_x)), s_1) \quad \text{où} \quad \mathcal{E}[\![x]\!]s = (v_x, s_1)$$

À noter ici que ces opérations ne sont définies que si les ensemble sont non vides. L'application de l'opération **`min`** ou **`max`** sur un ensemble vide mènera à une erreur d'exécution du programme.

4.3.2 Sémantique des expressions booléennes

À nouveau, la sémantique des constantes `true` et `false` est triviale :

$$\begin{aligned} \mathcal{E}[\![\textbf{true}]\!]s &= (\text{ true }, s) \\ \mathcal{E}[\![\textbf{false}]\!]s &= (\text{ false }, s) \end{aligned}$$

La sémantique des opérateurs booléens est définie comme suit :

$$\begin{aligned} \mathcal{E}[\![x \ \&\& \ y]\!]s &= (\text{ false }, s_1) \text{ si } v_x = \text{ false} \quad \text{où} \quad \mathcal{E}[\![x]\!]s = (v_x, s_1) \\ &\quad\ (v_y, s_2) \text{ si } v_x = \text{ true} \qquad\quad\ \ \mathcal{E}[\![y]\!]s_1 = (v_y, s_2) \\ \mathcal{E}[\![x \ || \ y]\!]s &= (\text{ true }, s_1) \text{ si } v_x = \text{ true} \quad \text{où} \quad \mathcal{E}[\![x]\!]s = (v_x, s_1) \\ &\quad\ (v_y, s_2) \text{ si } v_x = \text{ false} \qquad\quad\ \ \mathcal{E}[\![y]\!]s_1 = (v_y, s_2) \\ \mathcal{E}[\![!x]\!]s &= (\text{non } v_x, s_1) \qquad\qquad\qquad\quad \text{où} \quad \mathcal{E}[\![x]\!]s = (v_x, s_1) \end{aligned}$$

Remarquez bien ici que l'évaluation est opérandes pour le `&&` et le `||` est *paresseuse* (ou *lazy*), ce qui signifie que l'évaluation de la seconde opérande dépend du résultat de l'évaluation de la première. En d'autres mots : « Si on connait de résultat global de l'opération après l'évaluation de l'opérande de gauche, on n'évalue pas l'opérande de droite ». Après l'évaluation de l'opération, le programme pourra donc soit se trouver dans l'état s_1, soit dans l'état s_2.

La sémantique des opérateurs de comparaison est donnée comme suit :

$$\begin{aligned} \mathcal{E}[\![x <= y]\!]s &= (v_x \leq v_y, s_2) \quad \text{où} \quad \mathcal{E}[\![x]\!]s = (v_x, s_1) \\ &\qquad\qquad\qquad\quad\ \ \mathcal{E}[\![y]\!]s_1 = (v_y, s_2) \\ \mathcal{E}[\![x = y]\!]s &= (v_x = v_y, s_2) \quad \text{où} \quad \mathcal{E}[\![x]\!]s = (v_x, s_1) \\ &\qquad\qquad\qquad\quad\ \ \mathcal{E}[\![y]\!]s_1 = (v_y, s_2) \\ \mathcal{E}[\![x < y]\!]s &= (v_x < v_y, s_2) \quad \text{où} \quad \mathcal{E}[\![x]\!]s = (v_x, s_1) \\ &\qquad\qquad\qquad\quad\ \ \mathcal{E}[\![y]\!]s_1 = (v_y, s_2) \end{aligned}$$

Comme pour les opérations sur les entiers, l'évaluation se fait de gauche à droite.

La dernière opération booléenne est celle de test de la présence d'une valeur entière dans un ensemble d'entiers. Sa sémantique est la suivante :

$$\begin{array}{rcll} \mathcal{E}[\![x \ \texttt{in}\ y]\!]s & = & (\text{ true }, s_2) \text{ si } v_x \in v_y & \text{où} \quad \mathcal{E}[\![x]\!]s = (v_x, s_1) \\ & & (\text{ false }, s_2) \text{ sinon} & \quad\ \ \mathcal{E}[\![y]\!]s_1 = (v_y, s_2) \end{array}$$

4.3.3 Sémantique des appels de fonctions

Pour évaluer l'appel à une fonction, nous retrouvons la sémantique de cette fonction, dans l'environnement courant. Ensuite, les paramètres effectifs de la fonction sont évalués de gauche à droite. Enfin, si tout s'est bien passé, nous calculons le résultat dans l'état après évaluation des paramètres.

$$\mathcal{E}[\![x\ (\ y_1\ ,\ \ldots\ ,\ y_n\ )\ ]\!]_{(\sigma,\epsilon,i,o)} = f\, s_n\, (v_1 \ldots v_n)$$

$$\text{où} \left[\begin{array}{rcl} \epsilon(x,\ \text{func}\ ) & = & f \\ s_0 & = & (\sigma, \epsilon, i, o) \\ (v_i, s_i) & = & \mathcal{E}[\![y_1]\!]s_{i-1}, \quad (i = 1, \ldots, n) \end{array} \right.$$

Pour rappel, un état s est défini comme étant un quadruplet (σ, ϵ, i, o) où $\sigma \in \mathbb{S}$, $\epsilon \in \mathbb{E}$, $i \in \mathbb{I}$ et $o \in \mathbb{O}$. En d'autres termes :

- σ est une fonction partielle $\sigma : \mathbb{A} \hookrightarrow \mathbb{V} \cup \{\ \text{Uninit}\ \}$, qui à une adresse donnée *peut* renvoyer une valeur (ou uninit pour signaler que la valeur n'a pas été initialisée).
- ϵ est une fonction partielle $\epsilon : \mathsf{Id} \times \mathbb{C}\mathit{ategorie} \hookrightarrow \mathbb{F} \cup \mathbb{A} \cup \mathbb{T}\mathit{ypes}$, qui à un identifiant et un type de catégorie donné, *peut* renvoyer un environnement (qui correspond à la fonction appelée dans ce cas-ci).
- i représente les inputs.
- o représente les outputs.

La sémantique d'un appel de fonction sera donc l'application de f (l'environnement correspondant à la fonction x) dans un état de programme s_n (résultant de l'évaluation de chacun des arguments de la fonction de gauche à droite) avec comme arguments les résultats des évaluations des paramètres.

4.3.4 Sémantique des expressions gauches

La sémantique d'une expression gauche est la suivante :

$$\begin{array}{rcll} \mathcal{E}[\![x]\!]s & = & (\sigma_1(v_x), (\sigma_1, \epsilon_1, i_1, o_1)) & \text{où} \quad x \in \mathsf{ExprG} \\ & & & \quad\ \ \mathcal{A}[\![x]\!]s = (v_x, (\sigma_1, \epsilon_1, i_1, o_1)) \end{array}$$

Pour rappel, ϵ est une fonction partielle qui à une adresse donnée *peut* renvoyer une valeur. La fonction sémantique $\mathcal{A}$ donne, pour une expression gauche et un état donné, l'adresse en mémoire à laquelle la valeur courante de l'expression gauche est stockée. La sémantique de l'évaluation d'une expression gauche (dans une expression droite) correspond donc à la valeur stockée en mémoire pour l'expression gauche.

5 Expressions gauches

5.1 Syntaxe

Une expression gauche sert à référencer une variable. Elle est définie par la grammaire :

$$ExprG \quad \rightarrow \quad Id \tag{4}$$

5.2 Vérification des types

Nous pouvons maintenant détailler la fonction $\mathcal{T}ype_G$:

$$\mathcal{T}ype_G : ((\mathsf{Id} \times \mathbb{C}ategorie\ ) \rightarrow \mathbb{T}ypes\ ) \rightarrow (\mathsf{Id} \rightarrow \mathbb{T}ypes^{\,*}) \rightarrow \mathsf{ExprG} \hookrightarrow \mathbb{T}ypes$$

Une expression gauche sera correctement typée si :

$$\mathcal{T}ype_G\ ty\ sig\ [\![x]\!] \quad = \quad ty(x,\ \text{vp}\ ) \quad \text{où} \quad x \in \mathsf{Id}$$

Attention, $\mathcal{T}ype_G$ n'est définie que si ty et sig sont également définies, pour un identifiant donné. Cela signifie, intuitivement, que l'identifiant considéré *doit* avoir été déclaré dans le contexte où il est utilisé. Nous verrons plus loin que cela n'est possible que si l'identifiant est :

1. Un paramètre de la fonction en cours
2. Une variable locale à la fonction en cours
3. Une variable globale

5.3 Sémantique

La sémantique d'une expression gauche est donnée par la fonction :

$$\mathcal{A} : \mathsf{ExprG} \rightarrow \mathbb{S}tate \hookrightarrow \mathbb{A}$$

L'adresse d'une variable simple est obtenue en regardant dans l'environnement :

$$\mathcal{A}[\![x]\!](\sigma, \varepsilon, i, o) \quad = \quad \varepsilon(x,\ \text{vp}\ ) \tag{5}$$

6 Instructions

6.1 Syntaxe

Les instructions sont données par la grammaire suivante :

$$\begin{aligned}
Instr \quad \rightarrow \quad & \texttt{;} \\
| \ & ExprD\ \texttt{;} \\
| \ & ExprG\ \texttt{:=}\ ExprD\ \texttt{;} \\
| \ & \texttt{if (}\ ExprD\ \texttt{) then}\ Instr^{\,*}[\,\texttt{else}\ Instr^{\,*}]\ \texttt{fi ;} \\
| \ & \texttt{while (}\ ExprD\ \texttt{) do}\ Instr^{\,*}\ \texttt{od ;} \\
| \ & \texttt{write}\ ExprD\ \texttt{;} \\
| \ & \texttt{read}\ ExprG\ \texttt{;} \\
| \ & \texttt{add}\ ExprEnt\ \texttt{to}\ ExprG\ \texttt{;} \\
| \ & \texttt{remove}\ ExprEnt\ \texttt{from}\ ExprG\ \texttt{;} \\
| \ & \texttt{return}\ ExprD\ \texttt{;}
\end{aligned}$$

6.2 Vérification des types

Nous imposons les contraintes de type suivantes. Une instruction est bien typée si, étant donnés un dictionnaire de typage ty et de signatures sig :

Expression droite : une expression droite e est correctement typée si $\mathcal{T}ype\ ty\ sig\ [\![e]\!]$ est définie.

Affectation : l'instruction x **:=** e est bien typée si

$$\mathcal{T}ype\ ty\ sig[\![e]\!] = \mathcal{T}ype\ ty\ sig[\![x]\!]$$

et

$$\mathcal{T}ype\ ty\ sig[\![e]\!] \in \{\ \text{Int}\ ,\ \text{Bool}\ \}.$$

L'affectation est donc permise sur les **types scalaires (booléens et entiers) uniquement** !

Branchement conditionnel : l'instruction **`if`** **`(`** e **`)`** **`then`** i[**`else`** j] **`fi`** est correctement typée si

$$\mathcal{T}ype\ ty\ sig[\![e]\!] = \text{Bool}$$

et si ses blocs composants i et j sont aussi correctement typés.

Boucle while : l'instruction **`while`** (e) **`do`** i **`od`** est correctement typée si

$$\mathcal{T}ype\ ty\ sig[\![e]\!] = \text{Bool}$$

et si i est aussi bien typé.

Lecture : l'instruction **`read`** e est bien typée si

$$\mathcal{T}ype_G\ ty\ sig\ [\![e]\!] = \text{Int}$$

Ecriture : l'instruction **`write`** e est bien typée si

$$\mathcal{T}ype\ ty\ sig\ [\![e]\!] = \text{Int}$$

Ajout d'un élément à un ensemble : l'instruction **`add`** e **`to`** x est bien typée si

$$\mathcal{T}ype\ ty\ sig\ [\![x]\!] = \text{IntSet}$$

$$\mathcal{T}ype\ ty\ sig\ [\![e]\!] = \text{Int}$$

Retrait d'un élément à un ensemble : l'instruction **`remove`** e **`from`** x est bien typée si

$$\mathcal{T}ype\ ty\ sig\ [\![x]\!] = \text{IntSet}$$

$$\mathcal{T}ype\ ty\ sig\ [\![e]\!] = \text{Int}$$

Retour de résultat : l'instruction **`return`** e est bien typée si, en supposant qu'elle apparaisse dans le contexte d'une fonction x,

$$\mathcal{T}ype\ ty\ sig\ [\![e]\!] = ty(x,\ \text{func}\ )$$

En d'autres mots, l'instruction return est bien typée si le type de e est le même que le type de la fonction (i.e. le type de retour déclaré pour la fonction).

6.3 Sémantique

La sémantique des instructions est donnée par la fonction $\mathcal{S}$, déclarée comme

$$\mathcal{S} : \mathsf{Instr} \rightarrow \ \mathbb{S}tate \ \hookrightarrow (\ \mathbb{R}es \ \cup \ \mathbb{S}tate \ ).$$

Une instruction peut donc soit délivrer un résultat, soit délivrer un état intermédiaire. La définition de cette fonction est inductive, sur la structure des instructions, comme d'habitude.

6.3.1 Expression droite

Une expression droite est considérée comme une instruction. On jette la valeur de l'expression et on ne conserve que l'état s_1 résultant de son évaluation.

$$\mathcal{S}[\![e]\!]s = s_1 \text{ où } e \in \mathsf{ExprD} \text{ et } \mathcal{E}[\![e]\!]s = (v_e, s_1) \tag{6}$$

6.3.2 Affectation

L'affectation d'une expression droite à une expression gauche procède en deux étapes. D'abord, l'expression droite et l'expression gauche sont évaluées dans l'état courant. Ensuite, si ces évaluations donnent un résultat, on remplace la valeur stockée en mémoire à l'adresse calculée, par la valeur de l'expression droite.

On commence par évaluer l'expression gauche, ensuite on évalue l'expression droite :

$$\mathcal{S}[\![x \texttt{ := } e]\!](\sigma, \epsilon, i, o) = (\sigma_2[a/v_e], \epsilon_2, i_2, o_2) \tag{7}$$

où

$$\begin{aligned} (a, s_1) &= \mathcal{A}[\![x]\!](\sigma, \epsilon, i, o) \\ (v_e, (\sigma_2, \epsilon_2, i_2, o_2)) &= \mathcal{E}[\![e]\!]s_1 \end{aligned}$$

Dans l'équation ci-dessus, nous avons utilisé la notation $f[x/y]$ qui définit une nouvelle fonction g à partir de la fonction f de la façon suivante :

$$g(z) = \begin{cases} f(z) & \text{si } z \neq x \\ y & \text{si } z = x \end{cases}$$

A partir de cette définition, on voit que seule la cellule mémoire dont l'adresse a est spécifiée dans la partie gauche de l'expression voit son contenu remplacé par la nouvelle valeur v_e.

Pour rappel, un état s est définit comme étant un quadruplet (σ, ϵ, i, o), où σ est une fonction partielle qui à une adresse donnée *peut* renvoyer une valeur et ϵ est une fonction partielle qui à un identifiant et un type de catégorie donné, *peut* renvoyer un environnement.

6.3.3 Choix conditionnel

Le choix conditionnel **`if ( e ) then`** i_1 **`else`** i_2 **`fi`** signifie que si, e est évalué à true dans l'état actuel s, alors, i_1 doit s'exécuter. Si, au contraire, e est évalué à false , alors, i_2 doit s'exécuter. Il est possible que l'évaluation de e ne soit pas définie. Dans ce cas, l'ensemble de l'instruction n'est pas non plus définie.

$$\mathcal{S}[\![\texttt{ if (} e \texttt{) then } i_1 \texttt{ else } i_2 \texttt{ fi }]\!]s = \begin{cases} \mathcal{S}[\![st_1]\!]s & \text{si } \mathcal{E}[\![e]\!]s = \text{ true} \\ \mathcal{S}[\![st_2]\!]s & \text{si } \mathcal{E}[\![e]\!]s = \text{ false} \end{cases} \tag{8}$$

$$\mathcal{S}[\![\texttt{ if (} e \texttt{) then } i_1 \texttt{ fi }]\!]s = \mathcal{S}[\![\texttt{ if (} e \texttt{) then } i_1 \texttt{ else fi }]\!]s$$

6.3.4 Lecture sur le flux d'entrée

La lecture sur le flux d'entrée remplace la valeur stockée dans une cellule mémoire, dont l'adresse est passée en paramètre à l'instruction, par la première valeur apparaissant sur le flux d'entrée.

Deux situations peuvent amener à un échec [5] de l'exécution de cette instruction :

1. Il est impossible d'évaluer l'adresse passée en paramètre.
2. Le flux d'entrée est vide.

$$\mathcal{S}[\![\texttt{read } e \texttt{ ; }]\!]s = (\sigma_1[a/n], \epsilon_1, i_1, o_1) \text{ où } (a, (\sigma_1, \epsilon_1, n \cdot i_1, o_1)) = \mathcal{A}[\![e]\!]s \tag{9}$$

6.3.5 Écriture sur le flux de sortie

L'écriture sur le flux de sortie ajoute une valeur en queue de la file de sortie. Cette instruction peut échouer si l'évaluation de l'expression entière à écrire sur le flux de sortie n'est pas définie dans l'état courant.

$$\mathcal{S}[\![\texttt{write } e \texttt{ ; }]\!]s = (\sigma_1, \epsilon_1, i_1, o_1 \cdot v) \text{ où } (v, (\sigma_1, \epsilon_1, i_1, o_1)) = \mathcal{E}[\![e]\!]s \tag{10}$$

6.3.6 Retour de valeur

Le retour d'une valeur est la seule instruction dont la sémantique est un résultat (et non un état).

$$\mathcal{S}[\![\texttt{return } e \texttt{ ; }]\!]s = (v_e, s_1) \text{ où } \mathcal{E}[\![e]\!]s = (v_e, s_1) \tag{11}$$

6.3.7 Itération

L'opération d'itération est un petit peu plus difficile à définir. Nous allons procéder en deux temps. D'abord, nous donnerons la sémantique intuitive mais non-dénotationnelle et, ensuite, la sémantique dénotationnelle de la boucle.

Intuitivement, l'itération a la signification suivante : dans l'état de départ, on évalue la condition de séjour e, ce qui fournit un état intermédiaire s_1. Si l'évaluation de e est vraie, alors, on ne passe pas dans le corps de la boucle ; la sémantique de la boucle est donc s_1. Sinon, on exécute le corps de la boucle, qui est un bloc, en s_1. On obtient alors un élément s_2 qui peut être soit un résultat soit un état. Intuitivement, si s_2 est un résultat, cela signifie qu'une instruction **return** a été rencontrée pendant l'exécution du corps de la boucle. Dans ce cas, on considère que la boucle est terminée, son résultat étant s_2. Sinon, l'exécution du bloc se termine dans un état intermédiaire et l'on recommence à la première étape, à partir de ce nouvel état.

$$\mathcal{S}[\![\texttt{while(} e \texttt{) do } i \texttt{ od }]\!]s = \left(\begin{array}{ll} \underline{\text{soit}} & \mathcal{E}[\![e]\!]s = (x, s_1) \\ \underline{\text{si}} & x = \text{ false} \\ \underline{\text{alors}} & s_1 \\ \underline{\text{sinon}} & \left(\begin{array}{ll} \underline{\text{soit}} & s_2 = \mathcal{B}loc[\![i]\!]s_1 \\ \underline{\text{si}} & s_2 \in \mathbb{R}es \\ \underline{\text{alors}} & s_2 \\ \underline{\text{sinon}} & \mathcal{S}[\![\texttt{while(} e \texttt{) do } i \texttt{ od }]\!]s_2 \end{array} \right) \end{array} \right) \tag{12}$$

5. Il s'agit bien d'un *échec* et non d'une suspension, car nous avons supposé que le flux d'entrée était un fichier, contenant tous les inputs dont le programme aura besoin lors de son exécution.

6.3.8 Ajout d'un élément à un ensemble

L'instruction d'ajout d'un élément à un ensemble va d'abord évaluer l'expression droite (la valeur à ajouter) avant dévaluer l'expression gauche (l'ensemble dans lequel ajouter la valeur).

$$\mathcal{S}[\![\textbf{add } e \textbf{ to } x]\!](\sigma, \epsilon, i, o) = (\sigma_2[a/(v_x \cup \{v_e\})], \epsilon_2, i_2, o_2) \tag{13}$$

où

$$\begin{aligned} (v_e, s_1) &= \mathcal{E}[\![e]\!]s \\ (a, (\sigma_2, \epsilon_2, i_2, o_2)) &= \mathcal{A}[\![x]\!]s_1 \\ v_x &= \sigma_2 a \end{aligned}$$

L'expression droite e est évaluée dans l'état courant s pour donner la valeur v_e à ajouter à l'ensemble. L'expression gauche x est ensuite évaluée dans le nouvel état $s = (\sigma_1, \epsilon_1, i_1, o_1)$ afin d'obtenir l'adresse a de x. La nouvelle valeur de l'ensemble à l'adresse a est donnée par l'union de la valeur actuelle v_x (donnée grâce à l'application de la fonction σ_2 sur a) et de l'ensemble $\{v_e\}$.

6.3.9 Retrait d'un élément à un ensemble

Même chose que pour l'ajout d'un élément dans l'ensemble. La sémantique devient :

$$\mathcal{S}[\![\textbf{remove } e \textbf{ from } x]\!](\sigma, \epsilon, i, o) = (\sigma_2[a/(v_x \setminus \{v_e\})], \epsilon_2, i_2, o_2) \tag{14}$$

où

$$\begin{aligned} (v_e, s_1) &= \mathcal{E}[\![e]\!]s \\ (a, (\sigma_2, \epsilon_2, i_2, o_2)) &= \mathcal{A}[\![x]\!]s_1 \\ v_x &= \sigma_2 a \end{aligned}$$

Attention, selon la sémantique donnée ci-dessus, si v_e n'appartient pas à l'ensemble x, x restera inchangé.

7 Déclarations

Les déclarations de variables et de procédures mettent à jour le contexte courant (càd au-dessus de la pile de l'environnement courant), en assignant de nouvelles cellules mémoires aux identifiants déclarés.

7.1 Types de données

$$\mathit{Type} \quad \rightarrow \quad \textbf{int} \mid \textbf{bool} \mid \textbf{iset} \tag{15}$$

La sémantique d'un type est un élément de $\mathbb{T}ypes$:

$$\mathcal{T}ype : \mathsf{Type} \rightarrow \mathbb{T}ypes$$

$$\begin{aligned} \mathcal{T}ype[\![\textbf{bool}]\!] &= \text{Bool} \\ \mathcal{T}ype[\![\textbf{int}]\!] &= \text{Int} \\ \mathcal{T}ype[\![\textbf{iset}]\!] &= \text{IntSet} \end{aligned}$$

Nous définissons une fonction d'allocation :

$$\mathcal{A}lloc : \mathbb{T}ypes \rightarrow \mathbb{S} \rightarrow \mathbb{A} \times \mathbb{S} .$$

Pour rappel, $\mathbb{A}$ est l'ensemble des adresses et $\mathbb{S}$ est l'ensemble des mémoires. Cette fonction alloue l'espace mémoire nécessaire pour stocker un élément d'un certain type et renvoie l'adresse de base qui a été réservée.

$$\begin{array}{rcl} \mathcal{A}lloc[\![\texttt{bool}]\!]\sigma & = & (a, \sigma[a/Uninit]) \quad \underline{\text{si}}\ a \text{ est le plus petit naturel t.q. } \sigma(a) = \bot \\ \mathcal{A}lloc[\![\texttt{int}]\!]\sigma & = & (a, \sigma[a/Uninit]) \quad \underline{\text{si}}\ a \text{ est le plus petit naturel t.q. } \sigma(a) = \bot \\ \mathcal{A}lloc[\![\texttt{iset}]\!]\sigma & = & (a, \sigma[a/\emptyset]) \quad \underline{\text{si}}\ a \text{ est le plus petit naturel t.q. } \sigma(a) = \bot \end{array}$$

Attention, remarquez ici que la fonction d'allocation stocke la valeur *Uninit* dans l'espace alloué pour une variable booléenne ou entière. De la même manière, la fonction stocke *un ensemble vide* pour les variables de type ensemble d'entiers. Ceci signifie en pratique que lors de l'appel d'une fonction, tous les ensembles d'entiers déclarés localement dans cette fonction ont été initialisés *avant* l'exécution de la première instruction de la fonction. Cette remarque est reprise dans la description de la sémantique de l'appel d'une fonction au point 7.4.3.

7.2 Règles de nommage

Dans un programme LSD^{12}, les définitions suivantes sont interdites :

1. Une variable locale ne peut porter le même nom qu'une autre variable locale, dans la même fonction.
2. Une variable locale ne peut porter le même nom qu'un des paramètres formels de la fonction dans laquelle elle est définie.
3. Une variable locale peut porter le même nom que la fonction dans laquelle elle est définie.
4. Les paramètres formels d'une fonction peuvent porter le même nom que la fonction dans laquelle ils sont définis.
5. Une fonction peut porter le même nom qu'une variable.
6. Une fonction ne peut porter le même nom qu'une autre fonction de même niveau si la liste de leurs paramètres sont identiques. La surcharge de fonction est donc autorisée pour autant que leurs signatures[6] diffèrent. Trouver deux fonctions `fct(a:int):void;` et `fct(a:bool):int;` sur le même niveau est donc bien autorisé.

Ces règles permettent à une variable locale de porter le même nom qu'une variable d'un niveau supérieur. Dans ce cas, cette variable locale masque la variable globale. Une variable et une fonction définie dans un même contexte peuvent porter le même nom. Dans ce cas, le contexte syntaxique de l'utilisation permet de déterminer qui de la variable ou de la fonction est désignée.

7.3 Variables

7.3.1 Syntaxe

$$\textit{Decl\_Var} \quad \rightarrow \quad \textit{Id} \quad \textit{Type} \quad ;$$

6. pour rappel, la signature d'une fonction se compose de son nom et de la liste des types de ses paramètres. **Mais pas de son type de retour ! ! !**

7.3.2 Sémantique

Le type d'une variable sera donné par la fonction :

$$\mathcal{T}ype_V : \mathsf{DeclVar} \to ((\mathsf{Id} \times \mathbb{C}ategorie) \to \mathbb{T}ypes) \to ((\mathsf{Id} \times \mathbb{C}ategorie) \to \mathbb{T}ypes)$$

Une déclaration de variables met à jour un dictionnaire de types donné en assignant à chacun des identificateurs déjà déclarés un nouveau type.

$$\begin{array}{rcl} \mathcal{T}ype_V[\![\epsilon]\!]\ ty & = & ty \\ \mathcal{T}ype_V[\![(t\ x_1, \ldots, x_n\ ;) \cdot dl]\!]\ ty & = & (\mathcal{T}ype_V[\![dl]\!]ty)[(x_1,\ \text{vp})/\mathcal{T}ype[\![t]\!]] \ldots [(x_n,\ \text{vp})/\mathcal{T}ype[\![t]\!]] \\ & & \text{si t est dans } \mathsf{Type} \end{array}$$

La sémantique d'une déclaration de variable est donnée par la fonction :

$$\mathcal{D}_v : \mathsf{DeclVar} \to (\ \mathbb{S}tate \ \to \ \mathbb{S}tate\ )$$

$$\begin{array}{rcl} \mathcal{D}_v[\![\epsilon]\!]s & = & s \\ \mathcal{D}_v[\![(t\ x_1, \ldots, x_n\ ;) \cdot dl]\!]s & = & \mathcal{D}_v[\![(t\ x_1\ ;\ \ldots\ ;\ t\ x_n\ ;) \cdot dl]\!]s \\ \mathcal{D}_v[\![(t\ x\ ;) \cdot dl]\!](\sigma, \epsilon, i, o) & = & \mathcal{D}_v[\![dl]\!](\sigma_1, \epsilon_1[x/a], i, o) \\ & & \underline{\text{où}}\ (a, \sigma_1) = \mathcal{A}lloc[\![t]\!]\sigma \text{ si t est dans } \mathsf{Type} \end{array}$$

Dans la définition ci-dessus, nous avons utilisé l'opération $\epsilon[x/a]$, où ϵ est un environnement, càd une *pile de contextes* (de fonctions). Donc, si $\epsilon = e_1 \cdot e_2 \cdot \ldots \cdot e_n$, la mise à jour de ϵ est la mise à jour du contexte au-dessus de la pile : $e_1 \cdot \ldots \cdot e_n \cup \{x \mapsto a\}$.

7.4 Fonctions

7.4.1 Syntaxe

Function	→	*id* **(** **[** *List_Param*] **)** : *Type_Func* **;**	(16)
		Fct_Body | **forward;**	
Type_Func	→	**int** | **bool** | **void**	(17)
List_Param	→	(*List_Param* **,**)* *Decl_Param*	(18)
Decl_Param	→	[**var**] *Id* : *Type*	(19)
Fct_Body	→	*Decl_Bloc*	(20)
		Code_Bloc	
Decl_bloc	→	**var** (*Decl_Var* | *Function*)*	(21)
Code_Bloc	→	**begin** *Instr* * **end;**	(22)

Lorsque l'on déclare une fonction, on doit donc déclarer son type, son nom et ses paramètres formels. Les règles de nommage présentées ci-dessus sont d'application ici.

Une deuxième exigence se réfère au type des variables passées par valeur : un ensemble ne peut pas être passé passé par valeur, on ne peut le passer que par adresse.

Une troisième exigence se réfère à l'*ordre* de déclaration des fonctions : toute fonction doit avoir été déclarée (mais pas nécessairement définie) avant d'être appelée. Les appels "en avant" d'une

fonction définie après l'endroit de l'appel sont donc autorisé pour autant que la fonction appelée ait été déclarée **forward** en amont dans l'ordre des déclarations. Les seuls appels autorisés dans *Code_Bloc* sont donc :

1. les appels à une sous-fonction, déclarée dans *Decl_bloc* . A nouveau, seules les fonctions du plus haut niveau sont prises en compte. Les autres ne sont pas visibles à partir de f.
2. les appels récursifs à f.
3. les appels à une fonction déclarée "plus haut" que f.

Exemple d'appels valides et invalides : Le listing suivant présente différents cas d'appels autorisés et interdits :

```
program x;
 function main(): void
  var
     function fct1() : void; forward;

     function fct0() : int;
         var
             function fct01() : int;
                 var
                     function fct012() : int; var begin { ... } end;
                 begin
                     { ... }
                 end;

             function fct02() : int; var begin { ... } end;

         begin
             fct01(); { Autorisé }
             fct02(); { Autorisé }

             fct012(); { Interdit ! }

             fct0(); { Autorisé }

             fct1(); { Autorisé }

             fct2(); { Interdit ! }
         end;

     function fct1() : void; var begin { ... } end;
     function fct2() : void; var begin { ... } end;

 begin { ... } end;
end;
```

7.4.2 Vérification des types

Afin de faciliter l'écriture des règles de sémantique portant sur les fonctions et d'en augmenter la lisibilité, nous utiliserons dans la suite de cette section une notation "compacte à la Pascal" de la déclaration de fonction. La fonction LSD^{12} :

$$\begin{array}{ll} f & \textbf{(}\ [\ \textbf{var}\ ]t_1\ :\ p_1\ \textbf{,}\ \ldots\ \textbf{,}\ [\ \textbf{var}\ ]t_n\ :\ p_n]\ \textbf{)}\ :\ t\ \textbf{;} \\ & \textbf{var}\ \ dl \\ & \textbf{begin}\ \ il\ \ \textbf{end;} \end{array}$$

Sera donc notée :

$$t\ f\ \textbf{(}\ [\ \textbf{var}\ ]t_1\ p_1\ \textbf{,}\ \ldots\ \textbf{,}\ [\ \textbf{var}\ ]t_n\ p_n\ \textbf{)}\ \ \textbf{\{}\ dvl\ dfl\ il\ \textbf{\}}$$

Avec
– dvl la restriction de dl aux déclarations de variables.
– dfl la restriction de dl aux déclarations de fonction.

Et la fonction LSD^{12}, déclarée **forward** :

$$\begin{array}{ll} f & \textbf{(}\ [\ \textbf{var}\ ]t_1\ :\ p_1\ \textbf{,}\ \ldots\ \textbf{,}\ [\ \textbf{var}\ ]t_n\ :\ p_n]\ \textbf{)}\ :\ t\ \textbf{;} \\ & \textbf{forward;} \end{array}$$

Sera notée :

$$t\ f\ \textbf{(}\ [\ \textbf{var}\ ]t_1\ p_1\ \textbf{,}\ \ldots\ \textbf{,}\ [\ \textbf{var}\ ]t_n\ p_n\ \textbf{)}\ \ \textbf{\{}\ \varnothing\ \textbf{\}}$$

La mise à jour du dictionnaire des types, lorsque l'on a une fonction est effectuée en ajoutant les déclarations de variables (si il y en a) et de paramètres formels : [7]

$$\mathcal{T}ype[\![t\ f\ \textbf{(}\ [\ \textbf{var}\ ]t_1\ p_1\ \textbf{,}\ \ldots\ \textbf{,}\ [\ \textbf{var}\ ]t_n\ p_n\ \textbf{)}\ \ \textbf{\{}\ dvl\ il\ \textbf{\}}\ ]\!]ty = \mathcal{T}ype_V[\![t_1\ p_1\ \textbf{;}\ \ldots\ \textbf{;}\ t_n\ p_n\ \textbf{;}\ dvl]\!]ty$$

Étant donné une telle fonction, nous extrayons sa signature comme suit :

$$\mathcal{S}ig[\![t\ f\ \textbf{(}\ t_1\ p_1\ \textbf{,}\ \ldots\ \textbf{,}\ t_n\ p_n\ \textbf{)}\ \ \textbf{\{}\ dvl\ il\ \textbf{\}}\ ]\!] = (\mathcal{T}ype[\![t_1]\!], \ldots, \mathcal{T}ype[\![t_n]\!])$$

Une fonction est correctement typée, par rapport à un dictionnaire ty et sig si ses instructions il sont bien typées par rapport à sig et :

$$\mathcal{T}ype[\![t\ f\ \textbf{(}\ t_1\ p_1\ \textbf{,}\ \ldots\ \textbf{,}\ t_n\ p_n\ \textbf{)}\ \ \textbf{\{}\ dvl\ il\ \textbf{\}}\ ]\!]ty$$

7.4.3 Sémantique

La déclaration d'une fonction doit ajouter, dans l'état courant, la fonction sémantique qui y est associé. La sémantique d'une déclaration de fonction est calculée par la fonction :

$$\mathcal{D}_f : \mathsf{Function} \rightarrow (\ \mathbb{S}tate\ \rightarrow\ \mathbb{S}tate\ )$$

Qui est définie comme (où dpl signifie "déclaration de paramètres (liste)") :

7. Cette mise à jour ne dépendant pas du mode de passage nous nous permettons l'introduction de [**var**] pour signifier que tous les cas sont couvert par la présente définition

$$\mathcal{D}_f[\![tf\ \texttt{(}\ dpl\ \texttt{)}\ \ \texttt{\{}\ dvl\ il\ \texttt{\}}\ ]\!](\sigma,\epsilon,i,o) = (\sigma,\epsilon[f/\mathcal{F}[\![tf\ \texttt{(}\ dpl\ \texttt{)}\ \ \texttt{\{}\ dvl\ il\ \texttt{\}}\ ]\!],i,o). \tag{23}$$

Pour les fonction "totalement" définies et comme :

$$\mathcal{D}_f[\![tf\ \texttt{(}\ dpl\ \texttt{)}\ \ \texttt{\{}\ \varnothing\ \texttt{\}}\ ]\!](\sigma,\epsilon,i,o) = (\sigma,\epsilon[f/\mathcal{F}[\![tf\ \texttt{(}\ dpl\ \texttt{)}\ \ \texttt{\{}\ \varnothing\ \texttt{\}}\ ]\!],i,o). \tag{24}$$

Pour les fonction déclarées **`forward`** . Une fonction **`forward`** est donc une fonction "incomplète" au sens où seule sa signature est déclarée. Son implémentation *doit* être fournie par la suite.

Cette définition nous demande d'examiner comment sera définie la fonction sémantique associée à une fonction totalement définie tf **`(`** dpl **`)`** **`{`** $dvl\ il$ **`}`** par :

$$\mathcal{F} : \mathsf{Function} \rightarrow\ \mathbb{F}$$

Une fonction renvoie un résultat. Ce résultat dépend de deux éléments : l'état dans lequel la fonction est invoquée et les paramètres effectifs qui lui sont passés.

Son exécution peut être décomposée en cinq étapes :

- Création d'un environnement local, au-dessus de la pile des environnements,
- initialisation des cellules mémoires correspondant aux paramètres formels avec les valeurs ou les adresses des paramètres effectifs selon le type de passage.
- définition des variables locales (et initialisation des **`isets`** locaux)
- définition des fonctions locales
- exécution des instruction de la fonctions
- instruction **`return`** ou fin de fonction : suppression de l'environnement local, nettoyage de la mémoire et dans le cas d'un *return* retour de la valeur.

Remarque : **Nous ne demandons pas la vérification statique de la présence de l'instruction return**.

Supposons que nous disposions d'une fonction :

$$\mathcal{P}ar : \mathsf{List_{Param}} \rightarrow ((\mathsf{ExprD} \cup \mathsf{ExprG})^* \rightarrow (\ \mathbb{S}tate\ \hookrightarrow\ \mathbb{S}tate\ ))$$

Qui, lorsqu'on lui passe une liste de paramètres formels, retourne une fonction déterminant comment le passage des paramètres effectifs s'effectuera. Nous montrerons plus loin comment définir cette fonction.

Avant de définir :

$$\mathcal{F} : \mathsf{Function} \rightarrow\ \mathbb{F}$$

Il nous faut encore donner quelques explications à propos de l'opération de nettoyage à mener à la fin de l'exécution de la fonction.

L'opération de nettoyage (*garbage collection*) s'assure simplement que chaque cellule mémoire occupée est liée à un des identifiants de l'environnement courant [8]. Précisément, on dit qu'une adresse a est référencée soit s'il y a un identifiant qui est lié à a dans l'environnement soit s'il existe une adresse mémoire a' qui est référencée et dont le contenu est a. Cette définition est récursive et peut se comprendre en termes de graphes : les noeuds sont les cellules mémoires utilisées et les identifiants de l'environnement. Il y a un arc entre un noeud n_1 et un noeud n_2 si n_1 est un identifiant qui est lié à l'adresse n_2 ou bien si n_1 est une cellule mémoire qui contient une adresse pointant sur n_2. Les cellules référencées sont celles qui sont accessibles à partir des identifiants.

8. L'opération de garbage collection est exactement ce qui se passe lorsque vous *dépilez* suite à un retour de fonction, en P-Code. Nous ne demandons pas de récupérer la mémoire allouée aux ISets sur le tas, uniquement les cases mémoires de la pile qui les référencent

Cette adresse mémoire a' contient donc une adresse qui pointe sur a. Toutes les cellules mémoires qui ne sont pas référencées sont supprimées de la mémoire. Si l'exécution des instructions de la fonction ne se terminent pas sur un résultat, la sémantique de la fonction n'est pas définie : la fonction s'est plantée. Nous ne demandons pas de vérifier statiquement que chaque fonction contienne un **return** .

$$\begin{array}{rcl} \mathcal{F}[\![t\ f\ (\ dal\ )\ \{\ dvl\ il\ \}\ ]\!]s & = & \lambda s.\,\lambda v_1,\ldots v_n.\,(v, gc(s_1)) \\ & & \underline{\text{si}}\ \mathcal{S}[\![il]\!]\mathcal{D}_f[\![dfl]\!]\mathcal{D}_{vl}[\![dvl]\!](\sigma_1,\epsilon_1,i,o) = (v,s_1) \in\ \mathbb{R}es \\ & = & \lambda s.\,\lambda v_1,\ldots v_n.\,gc(s_1) \\ & & \underline{\text{si}}\ \mathcal{S}[\![il]\!]\mathcal{D}_f[\![dfl]\!]\mathcal{D}_{vl}[\![dvl]\!](\sigma_1,\epsilon_1,i,o) = s_1 \in\ \mathbb{S}tate \end{array} \tag{25}$$

Où
- $s = (\sigma, \epsilon, i, o)$
- $(\sigma_1, \epsilon_1, i, o) = \mathcal{P}ar[\![dal]\!](v_1, \ldots, v_n)(\epsilon.(\lambda x.\ \bot), \sigma, i, o)$
- La fonction de garbage collection $gc :\ \mathbb{S}tate\ \rightarrow\ \mathbb{S}tate$ est définie comme expliqué ci-dessus :

$$gc((e_1 \ldots e_n, \sigma, i, o)) = (\epsilon, \lambda a.\underline{\text{si}}\ a \in R\ \underline{\text{alors}}\ \sigma(a)\ \underline{\text{sinon}}\ \bot, i, o)$$

où $R \subseteq\ \mathbb{A}$ est le plus petit ensemble tel que

1. Pour tout identifiant x, $\epsilon(x) \in R$,
2. Pour toute adresse a, si $\exists a' \in R : \sigma(a') = a$, alors $a \in R$.

Il nous reste à définir comment les paramètres sont passés. Lorsqu'il n'y a pas de paramètres, on ne fait rien :

$$\mathcal{P}ar[\![\epsilon]\!]\epsilon\ s = s \tag{26}$$

Lorsqu'un paramètre est passé par adresse, on modifie le contexte courant pour qu'il référence, sous l'identifiant du paramètre formel, l'adresse du paramètre effectif, *dans l'environnement où l'appel a été effectué* (et non dans le contexte modifié par la déclaration, bien sûr).

$$\mathcal{P}ar[\![al \cdot \{\ ,\ \} \cdot\ \textbf{var}\ t\ x]\!](pel \cdot pe)s = (\sigma_1, \epsilon_1[x/\mathcal{A}[\![pe]\!]s], i_1, o_1) \tag{27}$$

Où $pe \in\ \mathbb{A}$ et $\mathcal{D}_{arg}[\![al]\!]pel\ s = (\sigma_1, \epsilon_1, i_1, o_1)$.

Lorsqu'un paramètre est passé par valeur, on modifie le contexte courant, pour qu'il référence une nouvelle cellule mémoire, sous l'identifiant du paramètre formel. Cette cellule est initialisée avec la valeur du paramètre effectif, évalué dans l'environnement où l'appel à lieu.

$$\mathcal{P}ar[\![al \cdot \{\ ,\ \} \cdot t\ x]\!](pel \cdot pe)s = (\sigma_1[a/\mathcal{E}[\![pe]\!]s], \epsilon_1[x/a], i_1, o_1) \tag{28}$$

Où $pe \in\ \mathbb{V}$, $\mathcal{D}_{arg}[\![al]\!]pels = (\sigma_1, \epsilon_1, i_1, o_1)$ et l'adresse a est telle que $\nexists id : \mathsf{Id} : \epsilon_1(id,\ \text{vp}\ ) = a$.

8 Liste de fonctions

La définition de la sémantique d'une liste de fonction se limite à une simple itération sur une liste, où nous appliquons la fonction $\mathcal{D}_f$ à chaque élément.

$$\mathcal{D}_{fl}[\![\epsilon]\!]s = s \tag{29}$$

$$\mathcal{D}_{fl}[\![df \cdot dfl]\!]s = \mathcal{D}_{fl}[\![dfl]\!](\mathcal{D}_f[\![df]\!]s) \tag{30}$$

A noter que lorsqu'une fonction est déclarée **forward** et que sa définition est donnée par la suite, cette nouvelle définition viendra bien écraser la définition incomplète. *Une fonction* **forward** *qui ne serait pas redéfinie donnera en une erreur de compilation.*

9 Programme

9.1 Syntaxe

$$\textit{Program} \quad \rightarrow \quad \textbf{program}\ \textit{id}\ \textbf{;}\ \textit{Function}\ \textbf{end;} \tag{31}$$
$$\tag{32}$$

9.2 Sémantique

La sémantique d'un programme est donnée par la fonction suivante :

$$\mathcal{P} : \mathbb{I} \hookrightarrow \mathbb{O}$$

Qui est définie de la façon suivante, pour un flux d'entrée donné i et un programme $funcl$.

L'exécution du programme commencera toujours pas l'exécution de la (seule et unique) fonction racine *Function* . Cette fonction ne doit posséder aucun paramètres et ne peut renvoyer de valeur de retour (doit être déclarée void).

L'état de départ doit être tel que :

1. l'environnement ne contient qu'un seul contexte, qui est vide : $e_\perp$.
2. le flux de sortie doit être vide (ϵ), ce qui signifie que rien n'a encore été écrit dans le fichier de sortie.

Remarquez que nous ne posons pas d'hypothèse sur l'état de la mémoire. Dans l'équation (33), nous utilisons donc une variable σ_0 qui peut représenter *n'importe quelle* mémoire. Même si toutes les cellules de la mémoire contiennent une valeur, cela n'a pas d'importance. En effet, afin de savoir si une cellule est allouée ou non, nous ne regardons pas dans la mémoire mais dans l'environnement. [9] Ceci correspond assez à un mécanisme de *garbage collection* : lorsqu'une cellule mémoire n'est plus référencée par aucun identificateur, on considère qu'elle est libérée et peut à nouveau être utilisée lors d'une prochaine allocation.

Dans cet état initial, nous définissons les alias et les fonctions globales, nous exécutons la fonction principale **P** et nous jetons l'environnement, le fichier d'entrée et la mémoire, et nous contentons de retourner le flux de sortie, à condition que celui-ci existe.

$$\begin{aligned} \mathcal{P}[\![funcl]\!]i \quad = \quad & \begin{cases} o' & \text{si} \quad \mathcal{F}(\text{ P , func })(\sigma_1, \epsilon_1, i_1, o_1)() = (\sigma', \varepsilon', i', o') \\ \perp & \text{si} \quad \mathcal{F}(\text{ P , func })(\sigma_1, \epsilon_1, i_1, o_1)() = \perp \end{cases} \\ & \text{où } (\sigma_1, \epsilon_1, i_1, o_1) = \mathcal{D}_{fl}[\![funcl]\!](\sigma_0, e_\perp, i, \epsilon) \end{aligned} \tag{33}$$

Cette façon de faire peut-être considérée comme travaillant sur un "buffer". Les écritures sur le flux de sortie ne sont "officialisées" que lorsque le programme s'est correctement terminé. Donc, si le

9. A nouveau, remarquez que cela signifie que $\mathcal{P}$ est une *relation* et non une fonction. Le choix d'une mémoire particulière n'influe pas (ne doit pas influer) sur le résultat du programme.

programme diverge ("se plante") sur sa dernière instruction et que des éléments ont déjà été écrits sur le flux de sortie auparavant, le résultat global de l'exécution du programme est $\perp$ et non pas un ensemble partiel de résultats.

10 Commentaires

10.1 Syntaxe

Des commentaires peuvent être introduits dans du code LSD^{12}. Les commentaires sont placés entre **{** et **}** .

10.2 Sémantique

Un commentaire est simplement ignoré par l'analyseur lexical. Sa sémantique est donc complètement nulle, du point de vue de l'exécution du programme.

11 Exemple de programme LSD^{12}

```
program exemple;
  function main(): void;
   var
     { declaration de variables + fonctions }
     a bool;
     b int;
     ensA iset;
     ensB iset;

     { Vérifie que ensA et ensB sont identiques }
     function equals() : bool;
         forward;

     { Initilise ensA avec des valeurs saisies par l'utilisateur }
     function buildEnsA() : void;
         var
            x int;
         begin
            read x;
            while ( !(x = 0)) do
                add x to ensA;
                read x;
                od;
         end;

     { Initilise ensB avec des valeurs saisies par l'utilisateur }
     function buildEnsB() : void;
         var
            x int;
```

```
        begin
            read x;
            while ( !(x = 0)) do
                add x to ensB;
                read x;
                od;
        end;

    { Définition complète de equals }
    function equals() : bool;
        var
                tmpA iset;
                tmpB iset;
                a int;
                b int;
                equal bool;
        begin
            equal := #ensA = #ensB;
            { Vérifier que les élements sont les mêmes }
            while (equal && 0 < #ensA && 0 < #ensB) do
                a := min ensA;
                b := min ensB;
                equal := (a = b);
                if (equal) then
                        add a to tmpA;
                        add b to tmpB;
                        remove a from ensA;
                        remove b from ensB;
                fi;
            od;
            { Remettre les ensembles dans leur état d'origine }
            while (0 < #tmpA && 0 < #tmpB) do
                a := min tmpA;
                b := min tmpB;
                add a to ensA;
                add b to ensB;
                remove a from tmpA;
                remove b from tmpB;
            od;
            return equal;
        end;

    ens1 iset;
    ens2 iset;

begin
    buildEnsA();
    buildEnsB();
```

```
        if(equals()) then
                write 1;
        else
                write 0;
        fi;
    end;
end;
```