

UPMC/master/info/4I503 APS  
Notes de cours

P. MANOURY

Janvier 2018

# 1 APS0: noyau fonctionnel

Le langage *APS0* est essentiellement un *langage d'expressions*. On y manipule des valeurs entières et booléenne. Ces valeurs sont désignées par des symboles de constantes ou peuvent être obtenues par *application* d'opérateurs primitifs (les opérations arithmétiques et booléennes usuelles, les opérateurs de comparaison).

Le langage *APS0* est *pleinement fonctionnel* en ce sens que les expressions y désignent non seulement les valeurs arithmétiques ou booléennes, mais également des *fonctions*. En ce sens, pour le langage *APS0*, une fonction est une valeur comme une autre: elle peut être le paramètre d'une fonction, voire, le résultat d'un calcul.

Le langage *APS0* permet de *définir* des constantes ou des fonctions; c'est-à-dire, permet de *lier* à des *noms* (ou *identificateurs*) des valeurs, fonctionnelles ou non.

Légère exception à son caractère fonctionnel, le langage *APS0* contient une, et une seule, *instruction* qui a pour but de produire l'affichage d'une valeur (arithmétique).

## 1.1 Syntaxe

La syntaxe est définie par un *lexique* et une *grammaire*. Le lexique définit les *unités lexicales* du langage ou *lexèmes*. On y trouve les *symboles réservés* et les *mots clef* du langage (que l'on peut aussi appeler *mots réservés*; ainsi que les ensembles de lexèmes utilisés pour désigner les *constantes numériques* et les *identificateurs*.

Lorsque l'on définit la grammaire du langage, les unités lexicales sont considérées comme des *symboles terminaux* dans le formalisme de la définition des grammaires. Les règles de grammaires définissent des ensembles de suites d'unités lexicales comme la valeur de *symboles non terminaux*.

Pour présenter la syntaxe, lexique et grammaire, on utilise un certain nombre de *convention typographiques*. Les *symboles réservés* et les *mots clef* du langage sont indiqués en caractère **machine à écrire**. Les ensembles de lexèmes, comme les *constantes numériques* ou les *identificateurs* sont indiqués en caractères **sans serif**. Les *symboles non terminaux* de la définition de la grammaire sont indiqués en caractères PETITES CAPITALES.

**Lexique** Dans la définition du lexique, on trouve l'énumération d'un certain nombre de suites de caractères explicitement données (symboles réservés et mots clef) ou la définition d'un ensemble des suites de caractères en utilisant les opérations sur les suites de caractères caractérisant les *expressions rationnelles* (*regular expressions*). Ici, nous emprunterons le formalisme de **lex** pour décrire les expressions rationnelles, plus précisément, sa déclinaison dans l'outil **ocamllex**.

**Symboles réservés** [ ] ( ) ; , \* ->

**Mots clef** CONST FUN REC ECHO bool int true false not and or eq lt add sub mul div if

**Constantes numériques** num défini par (''?)['0'-'9']+

**Identificateurs** ident défini par ([a'-'z''A'-'Z'])([a'-'z''A'-'Z''0'-'9'])\*  
dont on exclut les mots clef.

Appelons

- oprim l'ensemble de mots clef: not and or eq lt add sub mul div
- tprim l'ensemble de mots clefs bool int
- num l'ensemble des constantes numériques
- ident l'ensemble des identificateurs

Pour être opérationnelle, c'est-à-dire, permettre de reconnaître et d'isoler les lexèmes dans un flux de caractères, la définition du lexique spécifie également quels sont les *caractères séparateurs*. Il s'agit, en général de l'espace, la tabulation, le passage à la ligne et le retour chariot.

**Grammaire** La grammaire définit le sous ensemble des suites de lexèmes que l'on veut retenir pour le langage; c'est-à-dire, définit l'ensemble des suites de lexèmes acceptables comme un *programme*.

Appelons PROG cet ensemble. On pose qu'il est constitué de l'ensemble des *suites de commandes* encloses entre les symboles [ et ]. Appelons CMDS l'ensemble des suites de commandes. On définit alors PROG de la manière suivante:

PROG ::= [ CMDS ]

Une *commande* est soit une *déclaration*, soit une *instruction*. Nous appelons DEC l'ensemble des déclarations et STAT l'ensemble des instructions. Les *suites de commandes* du langage APS0 sont des suites de déclarations et d'instructions séparées par le symbole ; (point virgule). On fixe que le dernier membre d'une suite de commandes du langage APS0 est toujours une instruction et qu'il n'y a pas de suite de commandes vide dans ce langage. On définit alors l'ensemble CMDS de la manière suivante:

CMDS ::= STAT  
           | DEC ; CMDS  
           | STAT ; CMDS

C'est une définition récursive: par exemple, une suite de commande est une déclaration suivi d'un point virgule, lui-même suivi d'une suite de commandes. Cette définition récursive est *bien fondée* car il existe un *cas de base* à la définition: la suite consistant en une seule instruction. Naturellement, cette définition sera effectivement bien fondée lorsque l'on aura défini STAT.

Une *déclaration* est soit la déclaration, en fait, la *définition* d'une constante, la définition d'une fonction, possiblement récursive. Les suites de lexèmes correspondant aux déclarations sont introduites par les mots clefs CONST, FUN et REC selon les cas possibles. Une déclaration de constantes doit comprendre la mention du nom de la constante (un *identificateur*, élément de *ident*), le type de la constante et sa valeur qui est donnée sous forme d'une *expression*. Une fonction est définie à l'aide des mêmes éléments auxquels on ajoute la liste des paramètres formels de la fonction, avec leur type.

Un *type* est soit le symbole d'un type de base (élément de *tprim*), soit un type fonctionnel qui indique la suite des types des arguments de la fonction et le type du résultat de l'application de la fonction. La suite de type des arguments et le type du résultat sont séparés par le symbole ->; les types des arguments sont séparés par le symbole \*. Appelons TYPE l'ensemble des types pour APS0 et TYPES l'ensemble des suites de types pour les arguments. On définit de manière *mutuellement récursives* ces deux ensembles de la manière suivante:

TYPE ::= tprim  
           | ( TYPES -> TYPE )  
 TYPES ::= TYPE  
           | TYPE \* TYPES

Cette double définition est *bien fondée* car (intuitivement) la 1ère clause de la définition l'ensemble TYPE est un cas de base (*tprim*), donc la définition de TYPE est bien fondée; et alors la première clause de la définition de TYPES en constitue également un cas de base.

Dans une déclaration de fonction, un *argument* indique le nom de l'argument (identificateur, élément de *ident*) ainsi que son type (élément de TYPE). Appelons ARG l'ensemble de telles paires. On le définit par:

ARG ::= ident : TYPE

Dans une suite d'arguments, les paires indiquant le nom et le type des arguments sont séparés par le symbole , (virgule). Appelons ARGS l'ensemble de ces suites. Il est défini par:

ARGS ::= ARG  
           | ARG , ARGS

Appelons EXPR l'ensemble des *expressions* du langage APS0. Nous pouvons alors définir l'ensemble DEC de la manière suivante:

```

DEC ::=  CONST ident TYPE EXPR
      |  FUN ident TYPE [ ARGS ] EXPR
      |  FUN REC ident TYPE [ ARGS ] EXPR

```

L'ensemble STAT des instruction de *APSO* se définit en une seule clause:

```

STAT ::=  ECHO EXPR

```

Pour achever la définition de l'ensemble des programmes de *APSO*, reste à définir l'ensemble des expressions EXPR. Une expression du langage *APSO* est

- soit un symbole de constante booléenne (**true** ou **false**), soit un symbole de constante numérique (élément de **num**), soit un identificateur (élément de **ident**);
- soit *l'application* du mot clef **if** à trois expressions;
- soit *l'application* d'un symbole d'opération primitives (élément de **oprim**) à une *suite d'expressions*;
- soit *l'abstraction* fonctionnelle d'une expression obtenue en préfixant l'expression par une suite de déclaration d'arguments (élément de **ARGS**).

*L'application* est notée selon le mode *préfixé complètement parenthésé*. Les définitions de l'ensemble des expressions (EXPR et de l'ensemble des suites d'expressions, que nous appelons EXPRS sont données ainsi:

```

EXPR ::=  bool | num | ident
        |  ( if EXPR EXPR EXPR )
        |  ( oprim EXPRS )
        |  [ ARGS ] EXPR
        |  ( EXPR EXPRS )
EXPRS ::=  EXPR
          |  EXPR EXPRS

```

Ce sont deux *définitions mutuellement récursives* dont la bonne fonction repose sur les cas de base de la définition de EXPR qui sont **bool**, **num** et **ident**.

*Quod errat faciendum*

## 1.2 Typage

Les programmes écrits en *APSO* manipulent des valeurs entières ou booléennes au moyen des opérateurs de base fournis par le langage. Ces opérateurs sont donnés pour réaliser les fonctions arithmétiques et logiques usuelles: addition, soustraction, multiplication, division, négation, disjonction, conjonction. La sémantique donnera ce sens aux symboles du langage que l'on souhaite associer à ces opérations. En tant que fonctions arithmétiques et logiques, elles possèdent un *domaine* de définition qui spécifie à quel ensemble de valeurs doivent appartenir ses arguments et un *codomaine* qui spécifie à quel ensemble appartient le résultat de leur application. Dans le langage, ces ensembles de valeurs correspondent aux *types* **bool** et **int**. L'analyse de type d'une expression consiste à déterminer si l'application des opérateurs est conforme à la spécification du domaine des fonctions correspondantes. Si tel est le cas, l'analyse de type permet également de déterminer le type du résultat de l'application et celui-ci doit correspondre au codomaine de la fonction associée à l'opérateur.

Cela suppose, bien entendu que nous connaissions le type des opérateurs de base, ce qui est le cas. Toutefois, les identificateurs utilisés comme constantes symboliques, noms de fonctions ou noms de paramètres n'appartiennent *a priori* à aucun type en propre mais les déclarations de constantes (*cf* DEC) et de arguments formels (*cf* ARG) sont là pour leur en assigner un. L'ensemble des *assignments de types* à ces identificateurs et aux opérateurs est appelé *contexte de typage*.

La vérification de type appliquée aux expressions permet de sélectionner un sous-ensemble des éléments de EXPR qui *ont un sens* dans la perspective de leur évaluation. Ainsi, l'addition d'une valeur numérique à une valeur booléenne n'a pas, dans notre conception de langage *APSO*, de sens; de même l'application de

l'opérateur d'addition à trois expressions, fussent-elle des expressions arithmétiques. Ainsi, l'analyse de type que nous envisageons va-t-elle rejeter les expressions `(add 0 true)` et `(add 1 2 3)`, qui sont *syntactiquement correctes* mais que nous ne soumettrons pas au processus d'évaluation.

Dans le même ordre d'idée, nous vérifierons que l'expression suivant l'instruction d'affichage `ECHO` est bien une expression arithmétique, c'est-à-dire, de type `int`.

Pour ce qui est des déclarations, l'analyse de type est chargée de vérifier la cohérence de la déclaration. Par exemple, on vérifiera que le type annoncé d'un symbole de constante est bien celui de l'expression qui lui est associée dans la déclaration. Si cette cohérence est vérifiée, le contexte de typage est enrichi avec l'assignation du type déclarée au symbole défini.

**Contexte de typage** Un contexte de typage est donc une association entre symboles (opérateurs, constantes booléennes ou identificateurs) et types. Posons  $\text{sym} = \text{bool} \cup \text{oprim} \cup \text{ident}$ . Formellement, on modélise l'association réalisée dans un contexte de typage par une *fonction partielle* de l'ensemble des symboles de  $\text{sym}$ , dans l'ensemble des types  $\text{TYPE}$ . Posons  $G = \text{sym} \rightarrow \text{TYPE}$ . Dans cette perspective, si  $\Gamma$  est un contexte, alors  $\Gamma(x)$  désigne le type associé à  $x$  par  $G$ , lorsqu'il est défini.

Lors de l'analyse de type d'un programme, le contexte de typage évolue: chaque déclaration introduit une nouvelle association entre identificateur et type.

On note  $\Gamma[x : t]$  l'extension de  $\Gamma$  avec la liaison  $x$  à  $t$ . C'est-à-dire la fonction de  $G$  telle que

- $\Gamma[x : t](x) = t$
- $\Gamma[x : t](y) = \Gamma(y)$  lorsque  $x$  et  $y$  sont des symboles différents.

Abréviation: on écrit  $\Gamma[x_1 : t_1; x_2 : t_2]$  comme abréviation de  $\Gamma[x_1 : t_1][x_2 : t_2]$ . Plus généralement, on écrit  $\Gamma[x_1 : t_1; \dots; x_n : t_n]$  pour  $\Gamma[x_1 : t_1] \dots [x_n : t_n]$ .

On définit un contexte particulier que l'on note  $\Gamma_0$  est qui donne le type des opérateurs primitifs et des deux constantes booléennes. On pose que:

$\Gamma_0(\text{true})$	=	<code>bool</code>
$\Gamma_0(\text{false})$	=	<code>bool</code>
$\Gamma_0(\text{not})$	=	<code>bool</code> $\rightarrow$ <code>bool</code>
$\Gamma_0(\text{and})$	=	<code>bool</code> * <code>bool</code> $\rightarrow$ <code>bool</code>
$\Gamma_0(\text{or})$	=	<code>bool</code> * <code>bool</code> $\rightarrow$ <code>bool</code>
$\Gamma_0(\text{eq})$	=	<code>int</code> * <code>int</code> $\rightarrow$ <code>bool</code>
$\Gamma_0(\text{lt})$	=	<code>int</code> * <code>int</code> $\rightarrow$ <code>bool</code>
$\Gamma_0(\text{add})$	=	<code>int</code> * <code>int</code> $\rightarrow$ <code>int</code>
$\Gamma_0(\text{sub})$	=	<code>int</code> * <code>int</code> $\rightarrow$ <code>int</code>
$\Gamma_0(\text{mul})$	=	<code>int</code> * <code>int</code> $\rightarrow$ <code>int</code>
$\Gamma_0(\text{div})$	=	<code>int</code> * <code>int</code> $\rightarrow$ <code>int</code>

et que  $\Gamma_0(x)$  n'est pas défini pour tout  $x$  de `ident`.

**Jugement de typage** L'objectif de l'analyse de type est d'émettre un *jugement de typage*, par exemple: « dans le contexte  $\Gamma$ , l'expression  $e$  est de type  $t$  ». Formellement, ce jugement est une *relation* entre un contexte  $\Gamma$ , une expression  $e$  et un type  $t$ . L'analyse de type d'un programme complet utilise plusieurs catégories de jugement de typage. Ces catégories suivent *grosso modo* les catégories syntaxiques du langage *APS0*: les expressions, les instructions, les déclarations et les suites de commandes.

L'analyse de type des expressions de *APS0* doit garantir la correction de l'utilisation des composants de l'expression vis-à-vis des types. Lorsque cette correction est vérifiée, on peut assigner un type à l'expression. Un jugement de typage pour les expressions de *APS0* est l'énoncé de cette assignation. Les jugements acceptables sont définis formellement par la relation notée  $\vdash_{\text{EXPR}}$  qui est un sous ensemble des triplets de  $G \times \text{EXPR} \times \text{TYPE}$ . On écrit les jugements de typages, pour les expressions, sous la forme:  $\Gamma \vdash_{\text{EXPR}} e : t$ .

Dans *APSO*, le type associé à l'instruction **ECHO** n'est pas un élément de **TYPE**. L'instruction ne roduit pas de valeur, elle a simplement une *effet*, nous lui assignons donc un type spécifique que nous notons *void*. La relation de typage des instruction est notée  $\vdash_{\text{STAT}}$ , c'est un sous ensemble de  $G \times \text{STAT} \times \{\text{void}\}$ . On écrit:  $\Gamma \vdash_{\text{STAT}} s : \text{void}$

Nous traitons les déclarations pour ce qu'elles sont: des déclarations. Dans cette perspective, l'analyse de type d'une déclaration n'associe pas un type à une déclaration, mais un nouveau contexte de typage dans lequel est introduit l'association entre l'identificateur déclaré et son type à condition naturellement que la cohérence entre le type déclaré et le type de l'expression utilisée dans la déclaration à été établi. La relation de typage pour les déclaration est notée  $\vdash_{\text{DEC}}$ , c'est un sous ensemble de  $G \times \text{DEC} \times G$ . On écrit:  $\Gamma \vdash_{\text{DEC}} d : \Gamma'$ .

Dans *APSO*, les suites de commandes se terminent toutes par l'instruction **ECHO**, elles ne produisent pas plus de valeur que l'instruction. Nous leur assignerons le type *void*. Pour des raisons de facilités techniques, nous ne traitons pas exactement les suites de commandes telles que définie dans la syntaxe du langage (**CMDS**), mais l'ensemble de ces suites complétées, en fin de suite, par une *commande vide* que l'on note  $\varepsilon$ . On note  $\text{CMDS}_\varepsilon$  l'ensemble de telles suites. Si  $cs$  est un élément de **CMDS** on lui associe dans  $\text{CMDS}_\varepsilon$  la suite complétée notée  $cs; \varepsilon$ . La relation de typage pour les suites de commandes est notée  $\vdash_{\text{CMDS}}$ , c'est un sous ensemble de  $G \times \text{CMDS}_\varepsilon \times G$ . On écrit:  $\Gamma \vdash_{\text{CMDS}} cs : \text{void}$

Un programme de *APSO* étant simplement une suite de commandes, on lui assignera également le type *void*. On note  $\vdash$  la relation de typage, c'est un sous ensemble de  $\text{PROG} \times \{\text{void}\}$ . On écrit:  $\vdash p : \text{void}$ .

En résumé, pour définir la *discipline de type* de *APSO*, nous devons définir les cinq relations suivantes:

1.  $\vdash$  dans  $\text{PROG} \times \{\text{void}\}$ , on écrit  $\vdash p : \text{void}$ .
2.  $\vdash_{\text{CMDS}}$  dans  $G \times \text{CMDS}_\varepsilon \times G$ , on écrit:  $\Gamma \vdash_{\text{CMDS}} cs : \text{void}$
3.  $\vdash_{\text{DEC}}$  dans  $G \times \text{DEC} \times G$ , on écrit:  $\Gamma \vdash_{\text{DEC}} d : \Gamma'$ .
4.  $\vdash_{\text{STAT}}$  dans  $G \times \text{STAT} \times \{\text{void}\}$ , on écrit:  $\Gamma \vdash_{\text{STAT}} s : \text{void}$
5.  $\vdash_{\text{EXPR}}$  dans  $G \times \text{EXPR} \times \text{TYPE}$ , on écrit:  $\Gamma \vdash_{\text{EXPR}} e : t$ .

Les définitions de ces relations énoncent les conditions à satisfaire pour que chaque relation soit satisfaite. Un programme  $p$  est dit *bien typé* s'il satisfait la relation  $\vdash$ , c'est-à-dire que l'on a pu vérifier que  $\vdash p : \text{void}$ .

ATTENTION: bien que l'on présente dans ce qui suit les *règles de typage* sous forme d'une conditionnelle (*si ... alors ...*), il faut les comprendre comme des équivalences (*... si et seulement si ...*).

### 1.2.1 Expressions

On définit la relation  $\vdash_{\text{EXPR}}$  selon les *cas de construction* des expressions. Les cas de construction des expressions sont donnés par les clauses des règles syntaxiques qui définissent les expressions.

Les constantes numériques sont décrétées de type **int**:

(NUM) si  $n \in \text{num}$  alors  $\Gamma \vdash_{\text{EXPR}} n : \text{int}$

Un symbole a le type que lui donne le contexte:

(SYM) si  $x \in \text{sym}$  et si  $\Gamma(x) = t$  alors  $\Gamma \vdash_{\text{EXPR}} x : t$

Une abstraction fonctionnelle  $[x_1 : t_1, \dots, x_n : t_n]e$  est correctement typée lorsque l'expression  $e$  est correctement typée dans un contexte où les arguments ont le type indiqué dans l'abstraction. Dans ce cas, on peut assigner son type à l'abstraction:

(ABS) si  $\Gamma[x_1 : t_1; \dots; x_n : t_n] \vdash_{\text{EXPR}} e : t$  alors  $\Gamma \vdash_{\text{EXPR}} [x_1 : t_1, \dots, x_n : t_n]e : t_1 * \dots * t_n \rightarrow t$

Une application est correctement typée si le type de l'expression en position de fonction dans l'application a un type cohérents avec celui des expressions en position d'arguments de l'application. Dans ce cas, on peut donner le type de l'application:

(APP) si  $\Gamma \vdash_{\text{EXPR}} e_1 : t_1, \dots$  si  $\Gamma \vdash_{\text{EXPR}} e_n : t_n$   
 et si  $\Gamma \vdash_{\text{EXPR}} e : t_1 * \dots * t_n \rightarrow t$   
 alors  $\Gamma \vdash_{\text{EXPR}} (e \ e_1 \dots e_n) : t$

Enfin, l'expression alternative (**if**) est correctement typée si son premier argument est de type **bool** et l'ont peut assigner un même type aux deux alternants. Le type de l'expression alternative est le type des alternants.

(IF) si  $\Gamma \vdash_{\text{EXPR}} e_1 : \text{bool}$ , si  $\Gamma \vdash_{\text{EXPR}} e_2 : t$  et si  $\Gamma \vdash_{\text{EXPR}} e_3 : t$   
 alors  $\Gamma \vdash_{\text{EXPR}} (\text{if } e_1 \ e_2 \ e_3) : t$

L'opérateur d'alternative est *polymorphe*: il a potentiellement une infinité de types. Toutefois, tout ceux-ci doivent respecter la forme  $(\text{bool} * t * t) \rightarrow t$  avec  $t \in \text{TYPE}$ .

### 1.2.2 Instruction

On vérifie simplement que l'expression dont on veut afficher la valeur est de type **init**. Dans ce cas, on assigne le type **void** à l'instruction d'affichage:

(ECHO) si  $\Gamma \vdash_{\text{EXPR}} e : \text{int}$  alors  $\Gamma \vdash_{\text{STAT}} (\text{ECHO } e) : \text{void}$

### 1.2.3 Déclarations

On énonce une règle pour chaque cas de déclaration: constante, fonction, fonction récursive.

Une déclaration de constante est bien typée lorsque l'on peut assigner à l'expression qui définit la constante, le type déclaré pour cette constante. Dans ce cas, on ajoute une nouvelle liaison au contexte de typage:

(CONST) si  $\Gamma \vdash_{\text{EXPR}} e : t$  alors  $\Gamma \vdash_{\text{DEC}} (\text{CONST } x \ t \ e) : \Gamma[x : t]$

Une déclaration de fonction est correctement typée lorsque le contexte enrichi des assignation de type déclarées des arguments de la fonction permettent d'assigner le type déclaré du résultat de la fonction à l'expression qui définit la fonction:

(FUN) si  $\Gamma[x_1 : t_1; \dots; x_n : t_n] \vdash_{\text{EXPR}} e : t$   
 alors  $\Gamma \vdash_{\text{DEC}} (\text{FUN } x \ t \ [x_1 : t_1, \dots, x_n : t_n] \ e) : \Gamma[x : t_1 * \dots * t_n \rightarrow t]$

Une déclaration des fonction récursive est correctement typée dans les même conditions qu'une fonction simple en enrichissant également le contexte de vérification de type de l'expression avec la liaison du nom de la fonction avec son type déclaré:

(FUNREC) si  $\Gamma[x_1 : t_1; \dots; x_n : t_n; x : t_1 * \dots * t_n \rightarrow t] \vdash_{\text{EXPR}} e : t$   
 alors  $\Gamma \vdash_{\text{DEC}} (\text{FUN REC } x \ t \ [x_1 : t_1, \dots, x_n : t_n] \ e) : \Gamma[x : t_1 * \dots * t_n \rightarrow t]$

### 1.2.4 Suites de commandes

On applique la vérification de type aux suites de commandes complétées par  $\varepsilon$ . On examine, récursivement, les éléments de la suite les uns après les autres, dans l'ordre donné par la suite. Lorsque la suite n'est pas réduite à  $\varepsilon$ , selon la forme du premier terme de la suite, on distingue le cas des déclarations de celui des instructions. Les suites de commandes doivent avoir le type **void**.

Si la suite commence par une déclaration et que celle-ci est bien typée, on poursuit la vérification sur le reste de la suite avec le contexte obtenu après vérification de la déclaration de tête:

(DECS) si  $d \in \text{DEC}$ , si  $\Gamma \vdash_{\text{DEC}} d : \Gamma'$  et si  $\Gamma' \vdash_{\text{CMDs}} cs : \text{void}$  alors  $\Gamma \vdash_{\text{CMDs}} (d; \ cs) : \text{void}$ .

Si la suite commence par une instruction, on vérifie que celle-ci est correctement typée, puis on vérifie le reste de la suite:

(STATS) si  $s \in \text{STAT}$ , si  $\Gamma \vdash_{\text{STAT}} s : \text{void}$  et si  $\Gamma \vdash_{\text{CMDs}} cs : \text{void}$  alors  $\Gamma \vdash_{\text{CMDs}} (s; cs) : \text{void}$ .

La suite vide est de type *void*

(END)  $\Gamma \vdash_{\text{CMDs}} \varepsilon : \text{void}$ .

### 1.2.5 Programmes

Un programme est correctement typé si la suite de commande qui le compose est correctement typée, dans le contexte initial  $\Gamma_0$ :

(PROG) si  $\Gamma_0 \vdash_{\text{CMDs}} (cs; \varepsilon) : \text{void}$  alors  $\vdash [cs] : \text{void}$

## 1.3 Sémantique

La sémantique d'un langage de programmation doit définir quelle est la valeur ou l'effet attendu de l'exécution des programmes de ce langage. Pour cela, la sémantique définit le comportement dynamique des différents composants d'un programme. Comme les règles de typage, elle est définie par *cas de construction* des programmes.

Autre analogie avec le typage, on définit la sémantique comme une relation entre un *contexte d'évaluation*, une construction syntaxique et une valeur ou un effet. Mais ici, les contextes d'évaluation associent une *valeur* aux identificateurs et non plus simplement un type.

Les valeurs manipulées lors de l'exécution d'un programme de *APSO* sont de trois sortes:

- les *valeurs immédiates* que nous pourrions limiter pour *APSO* au seul ensemble des entiers naturels  
Dans la sémantique, nous ne distinguons pas valeurs numériques des valeurs booléenne: la séparation de leur usage ayant été garantie par le typage.
- les valeurs que nous devons associer aux fonctions que nous appelons de *fermetures*.  
La sémantique que nous donnons à *APSO* assigne au langage de conserver une *liaison statique* des identificateurs. Par exemple, tous les identificateurs utilisés dans la définition d'une fonction, à l'exclusion de ses paramètres, conservent la même valeur à chaque appel de la fonction. Ainsi, une fermeture est constituée du code de la fonction et de l'environnement tel qu'il existe au *moment* de la définition de la fonction.
- les valeurs associées aux définitions récursives qui ne sont pas de simples fermetures, mais des fonctions de construction d'une fermeture.

Un autre élément à *modéliser* pour définir la sémantique des programmes de *APSO* est l'effet de l'instruction d'affichage: l'envoi ou l'ajout d'une valeur dans un *flux de sortie*. Anstraitement, un flux de sortie est simplement une suite de valeurs.

**Domaines sémantiques** Pour rendre les choses plus formelles, posons:

**Valeurs immédiates**  $N$

**Fermetures**  $F = \text{EXPR} \times (V^* \rightarrow E)$

**Fermetures récursives**  $FR = F \rightarrow F$

**Valeurs**  $V = N \oplus F \oplus FR$

**Environnement**  $E = \text{ident} \rightarrow V$  (fonction partielle)

**Flux de sortie**  $O = N^*$



L'ensemble des valeurs  $V$  est défini comme la *somme disjointe* des trois ensembles  $N$ ,  $F$  et  $FR$ . Une somme (ou union) disjointe, comme son nom l'indique, réuni en un seul ensemble les éléments de plusieurs ensemble, mais, à la différence de l'union simple, on sait, dans une somme disjointe distinguer l'origine des éléments de la somme.

Intuitivement, dans une somme disjointe, les éléments conservent une *marque* de leur origine. Formellement, cette marque est représentée par une *injection canonique*. Par exemple, pour l'ensemble  $V$  des valeurs, nous avons trois injections canoniques notées  $inN$ ,  $inF$  et  $inFR$  dont les domaines sont, respectivement,  $V$ ,  $F$  et  $FR$  et qui partagent le codomaine  $V$ . Ainsi, tous les éléments de  $V$  peuvent s'écrire sous l'une des formes:  $inN(n)$ ,  $inF(f)$  ou  $inFR(r)$  avec  $n \in N$ ,  $f \in F$  et  $r \in FR$ .

**Contextes d'évaluation** Un contexte d'évaluation est un couple formé d'un environnement (fonction dans  $E$ ) et d'un flux de sortie.

Comme les contextes de typage, les environnements évolueront au fur et à mesure de l'évaluation. Par exemple, une déclaration ajoute une liaison entre identificateur et valeur à l'environnement. Si  $\rho$  est un environnement,  $x$ , un identificateur et  $v$  une valeur, on note  $\rho[x = v]$  l'extension de l'environnement  $\rho$  avec la nouvelle liaison de  $x$  à  $v$ ;  $\rho[x = v]$  désigne la fonction de  $E$  telle que  $\rho[x = v](x) = v$  et  $\rho[x = v](y) = \rho(y)$  lorsque  $x$  et  $y$  sont des symboles différents. Si  $\omega$  est un flux de sortie, et  $n$  un entier, on note  $n;\omega$  l'ajout de  $n$  au flux de sortie.

**Fonctions utiles** La possibilité de faire réaliser des calculs par les programmes de *APSO*, comme pour tout autre langage de programmation, repose sur l'existence de *fonctions primitives* que les programmes peuvent invoquer. Dans les «vrais langages», ceux qui sont destinés à tourner sur des «vraies machines», ces fonctions primitives sont *implantées* dans les micro-processeurs par des circuits réalisant les opérations arithmétiques, pour les plus simples, ou du code assembleur pour les plus complexes. Nous ne descendrons pas jusqu'à ce niveau de détail et nous contenterons de supposer l'existence d'une fonction qui saura associer à chaque symbole d'opération primitive un calcul sur les valeurs entières. Appelons  $\pi$  cette fonction et posons:

$$\begin{aligned}
\pi(\text{not})(0) &= 1 \\
\pi(\text{not})(1) &= 0 \\
\pi(\text{and})(0, n) &= 0 \\
\pi(\text{and})(1, n) &= n \\
\pi(\text{or})(1, n) &= 1 \\
\pi(\text{or})(0, n) &= n \\
\pi(\text{eq})(n_1, n_2) &= 1 && \text{si } n_1 = n_2 \\
&= 0 && \text{sinon} \\
\pi(\text{lt})(n_1, n_2) &= 1 && \text{si } n_1 < n_2 \\
&= 0 && \text{sinon} \\
\pi(\text{add})(n_1, n_2) &= n_1 + n_2 \\
\pi(\text{sub})(n_1, n_2) &= n_1 - n_2 \\
\pi(\text{mul})(n_1, n_2) &= n_1 \cdot n_2 \\
\pi(\text{div})(n_1, n_2) &= n_1 \div n_2
\end{aligned}$$

Enfin, le code des programmes fera mention de constantes numériques: les unités lexicales de l'ensemble **num**. Les calculs ne sont pas effectués directement sur les symboles de ces constantes, mais sur les valeurs immédiates qu'ils représentent. Pour les «vraies machines» ces valeurs immédiates sont les codage binaire des nombres, pour nous ce sera plus simplement les éléments de  $N$ . Il nous faut donc un moyen de passer des éléments de **num** à ceux de  $N$ . Pour ce, on se donne une fonction  $\nu$  telle que, par exemple  $\nu(\text{trm42}) = 42$ .

**Relations sémantiques** On retrouve pour la sémantique cinq relations liant les contextes d'évaluation et les constructions syntaxique à leur résultat. Nous utiliserons les mêmes symboles que pour le typage, mais leur domaines (et la notation) seront différents.

L'évaluation d'un programme *APS0* produit un flux de sortie, la relation sémantique qui la définit associe un programme à un élément de  $O$ .

L'évaluation d'une déclaration produit un nouvel environnement, la relation sémantique qui la définit associe un environnement et une déclaration à un nouveau environnement.

L'évaluation de l'instruction produit un nouveau flux de sortie en utilisant la valeur obtenue par évaluation d'une expression, sa relation sémantique associe un environnement, un flux de sortie et une instruction à un nouveau flux de sortie.

L'évaluation d'une expression produit une valeur, sa relation sémantique associe un environnement et une expression à une valeur.

Enfin, l'évaluation d'une suite de commandes produit un nouveau flux de sortie, sa relation sémantique associe un environnement, un flux de sortie et une suite de commandes à un nouveau flux de sortie. Comme pour le tyage, on évalue des suites terminée par la commande vide  $\varepsilon$ .

Formellement

**Programme**  $\vdash$ , dans  $\text{PROG} \times O$ , on écrit  $\vdash p \rightsquigarrow \omega$ .

**Déclaration**  $\vdash_{\text{DEC}}$  dans  $E \times \text{DEC} \times E$ , on écrit  $\rho \vdash_{\text{DEC}} d : \rho'$ .

**Instruction**  $\vdash_{\text{STAT}}$  dans  $E \times O \times \text{STAT} \times O$ , on écrit  $\rho, \omega \vdash_{\text{STAT}} s \rightsquigarrow \omega$ .

**Expression**  $\vdash_{\text{EXPR}}$   $E \times \text{EXPR} \times V$ , on écrit  $\rho \vdash_{\text{EXPR}} e \rightsquigarrow v$ .

**Suite de commandes**  $\vdash_{\text{CMDs}}$  dans  $E \times O \times \text{CMDs}_\varepsilon \times O$ , on écrit  $\rho, \omega \vdash_{\text{CMDs}} cs \rightsquigarrow \omega'$ .

### 1.3.1 Expressions

Les valeurs des constantes booléennes **true** et **false** sont, respectivement les valeurs 1 et 0.

(TRUE)  $\rho \vdash_{\text{EXPR}} \mathbf{true} \rightsquigarrow \text{inN}(1)$

(FALSE)  $\rho \vdash_{\text{EXPR}} \mathbf{false} \rightsquigarrow \text{inN}(0)$

La valeur d'une constante numérique est donnée par la fonction  $\nu$ :

(NUM) si  $n \in \text{num}$  alors  $\rho \vdash_{\text{EXPR}} n \rightsquigarrow \text{inN}(\nu(n))$

La valeur d'un identificateur est celle donnée par l'environnement:

(ID) si  $x \in \text{ident}$  et  $\rho(x) = v$  alors  $\rho \vdash_{\text{EXPR}} x \rightsquigarrow v$

La valeur de l'application d'une fonction primitives est obtenue par appel aux fonctions fournies par la fonction  $\pi$ , après évaluation des arguments appliqués:

(PRIM) si  $x \in \text{oprim}$ , si  $\rho \vdash_{\text{EXPR}} e_1 \rightsquigarrow \text{inN}(n_1), \dots$ , si  $\rho \vdash_{\text{EXPR}} e_k \rightsquigarrow \text{inN}(n_k)$  et si  $\pi(x)(n_1, \dots, n_k) = n$  alors  $\rho \vdash_{\text{EXPR}} (x \ e_1 \dots e_n) \rightsquigarrow \text{inN}(n)$

La valeur d'une expression alternative dépend de la valeur de son premier argument:

(IF1) si  $\rho \vdash_{\text{EXPR}} e_1 \rightsquigarrow \text{inN}(1)$  et si  $\rho \vdash_{\text{EXPR}} e_2 \rightsquigarrow v$  alors  $\rho \vdash_{\text{EXPR}} (\mathbf{if} \ e_1 \ e_2 \ e_3) \rightsquigarrow v$

(IF0) si  $\rho \vdash_{\text{EXPR}} e_1 \rightsquigarrow \text{inN}(0)$  et si  $\rho \vdash_{\text{EXPR}} e_3 \rightsquigarrow v$  alors  $\rho \vdash_{\text{EXPR}} (\mathbf{if} \ e_1 \ e_2 \ e_3) \rightsquigarrow v$

Notez comment, dans une expression alternative seuls deux arguments sont évalués, et non trois.

La valeur d'une abstraction fonctionnelle est une fermeture

(ABS)  $\rho \vdash_{\text{EXPR}} [x_1:t_1, \dots, x_n:t_n]e \rightsquigarrow \text{inF}(e, \lambda v_1 \dots v_n. \rho[x_1 = v_1; \dots; x_n = v_n])$

La fermeture associée à l'abstraction  $[x_1:t_1, \dots, x_n:t_n]e$  est le couple formé de l'expression  $e$  qui est l'expression qu'il faudra évaluer lorsque la fonction sera appliquée et d'une fonction (notée  $\lambda v_1 \dots v_n. \rho[x_1 = v_1; \dots; x_n = v_n]$ ) qui construit l'extension de  $\rho$  avec des valeurs pour les paramètres formels  $(x_1 \dots x_n)$  de la fonction.

La valeur de l'application d'une expression (dont la valeur doit être une fermeture) à d'autres expressions est obtenue, après évaluation de ces dernières, par évaluation du premier terme de la fermeture (corps de la fonction) dans le contexte construit par extension du contexte mémorisé dans la fermeture:

(APP) si  $\rho \vdash_{\text{EXPR}} e \rightsquigarrow \text{inF}(e', r)$ , si  $\rho \vdash_{\text{EXPR}} e_1 \rightsquigarrow v_1, \dots$ , si  $\rho \vdash_{\text{EXPR}} e_n \rightsquigarrow v_n$   
 et si  $r(v_1, \dots, v_n) \vdash_{\text{EXPR}} e' \rightsquigarrow v$   
 alors  $\rho \vdash (e \ e_1 \dots e_n) \rightsquigarrow v$

Si la valeur de l'expression en position de fonction est une fermeture récursive, on applique cette fermeture à elle-même avant de procéder comme ci-dessus:

(APPR) si  $\rho \vdash_{\text{EXPR}} e \rightsquigarrow \text{inFR}(\varphi)$ , si  $\varphi(\text{inFR}(\varphi)) = \text{inF}(e', r)$ ,  
 si  $\rho \vdash_{\text{EXPR}} e_1 \rightsquigarrow v_1, \dots$ , si  $\rho \vdash_{\text{EXPR}} e_n \rightsquigarrow v_n$   
 et si  $r(v_1, \dots, v_n) \vdash_{\text{EXPR}} e' \rightsquigarrow v$   
 alors  $\rho \vdash (e \ e_1 \dots e_n) \rightsquigarrow v$

### 1.3.2 Instruction

L'évaluation de l'instruction d'affichage ajoute un entier au flux de sortie:

(ECHO) si  $\rho, \omega \vdash_{\text{EXPR}} e \rightsquigarrow \text{inN}(n)$  alors  $\rho, \omega \vdash_{\text{STAT}} \text{ECHO } e \rightsquigarrow (n; \omega)$

### 1.3.3 Déclarations

L'évaluation d'une déclaration de constante ajoute à l'environnement la liaison entre le nom de la constante et la valeur obtenue par évaluation de l'expression qui la définit:

(CONST) si  $\rho \vdash_{\text{EXPR}} e \rightsquigarrow v$  alors  $\rho \vdash_{\text{DEC}} (\text{CONST } x \ t \ e) \rightsquigarrow \rho[x = v]$

L'évaluation de la déclaration d'une fonction (non récursive) ajoute à l'environnement la liaison entre le nom de la fonction et la fermeture qui représente cette fonction:

(FUN)  $\rho \vdash_{\text{DEC}} (\text{FUN } x \ t \ [x_1:t_1, \dots, x_n:t_n] \ e) \rightsquigarrow \rho[x = \text{inF}(e, \lambda v_1 \dots v_n. \rho[x_1 = v_1; \dots; x_n = v_n])]$

L'évaluation de la déclaration d'une fonction récursive ajoute à l'environnement une liaison entre le nom de la fonction et la fermeture récursive qui la représente:

(FUNREC)  $\rho \vdash_{\text{DEC}} (\text{FUN REC } x \ t \ [x_1:t_1, \dots, x_n:t_n] \ e)$   
 $\rightsquigarrow \rho[x = \text{inFR}(\lambda f. \text{inF}(e, \lambda v_1 \dots v_n. \rho[x_1 = v_1; \dots; x_n = v_n][x = f])]$

### 1.3.4 Suite de commandes

On distingue le cas des suites commençant par une déclaration de celles commençant par une instruction.

Si la suite commence par une déclaration, l'évaluation de la suite est obtenue par l'évaluation du reste de la suite dans un environnement produit par la déclaration:

(DECS) si  $\rho, \omega \vdash_{\text{DEC}} d \rightsquigarrow \rho'$  et si  $\rho', \omega \vdash_{\text{CMDs}} cs \rightsquigarrow \omega'$  alors  $\rho, \omega \vdash_{\text{CMDs}} (d; \ cs) \rightsquigarrow \omega'$

Si la suite commence par une instruction, l'évaluation de la suite est obtenue par l'évaluation du reste de la suite dans un contexte où le flux de sortie a été modifié par l'évaluation de l'instruction:

(STATS) si  $\rho, \omega \vdash_{\text{STAT}} s \rightsquigarrow \omega'$  et si  $\rho, \omega' \vdash_{\text{CMDs}} cs \rightsquigarrow \omega''$  alors  $\rho, \omega \vdash_{\text{CMDs}} (s; \ cs) \rightsquigarrow \omega''$

L'évaluation de la suite vide laisse inchangé son contexte d'évaluation:

(END)  $\rho, \omega \vdash_{\text{CMDs}} \varepsilon \rightsquigarrow \omega$

### 1.3.5 Programme

L'évaluation d'un programme est l'évaluation de la suite de commandes (complétée par l'instruction vide  $\varepsilon$ ) qui le constitue avec un contexte initial vide noté  $\emptyset, \emptyset$ :

(PROG) si  $\emptyset, \emptyset \vdash_{\text{CMDS}} cs; \varepsilon \rightsquigarrow \omega$  alors  $\vdash [cs] \rightsquigarrow \omega$