

Projet METAL

Grammaire

Auteur : Sylvain Huet

Création : 13/01/03

Dernière mise-à-jour : 20/02/07

1 Introduction

1.1 *Aperçu sur le document*

Grammaire du langage Métal.

2 Description

2.1 *Types*

<i>Type</i>	=	<i>B</i>		<i>B(labels*)</i>
		un		wn
		rn		
		listType		tabType [<i>Type</i> *]
		fun [Type*] Type		
<i>TypeMono</i>	=	<i>B</i>		<i>B(labels*)</i>
		wn		rn
		listTypeMono		tabTypeMono [<i>TypeMono</i> *]
		fun [TypeMono*] TypeMono		
<i>B</i>	=	Type de base		
un	=	variable liée		
rn	=	réursion de niveau <i>n</i>		

Les types de base sont :

I	:	int
S	:	string
F	:	float
Env	:	environnement

Cette liste n'est pas exhaustive : grâce aux structures et aux constructeurs de types, vous pouvez vous-même développer vos types de base.

Quelques commentaires sur le tableau, si vous n'êtes pas familier avec ces notations.

La première ligne définit l'expression *Type*, qui est en fait le type Scol. Puis, séparés par des '|', on trouve les différentes manières d'écrire l'expression : cela peut être :

- *B*, qui est défini à la troisième ligne : c'est un type de base, tel que I, S, F, ... Donc puisque I est un type de base, *B* peut s'écrire I, et puisque *Type* peut s'écrire *B*, I est bien un type en Métal.

- **un** avec *n* entier : u0, u1, u2, u3, ... sont des types : ils correspondent aux variables liées.

- **wn** avec *n* entier : w0, w1, w2, w3, ... sont des types faibles.

- **rn** avec *n* entier : r0, r1, r2, r3, ... sont des types : ils définissent les récursions dans les types.

- **tab** *Type* : type tableau. Le mot **tab** est suivi du type des éléments du tableau. Par exemple `tab I` est le type d'un tableau d'entiers.

- **list** *Type* : type tableau. Le mot **list** est suivi du type des éléments de la liste. Par exemple `list I` est le type d'une liste d'entiers.

- [*Type**] : type tuple. Entre crochets, on écrit plusieurs *Types* (c'est le sens de l'étoile *). Par exemple, [*I S*] est un tuple de deux éléments dont le premier est un entier et le second une chaîne de caractères. Eventuellement, le tuple est vide : []. Le tuple peut lui-même contenir des tuples : [*I [S I]*]

- **fun** [*Type**] *Type* : type fonction. le mot **fun** est suivi d'un tuple contenant les arguments de la fonction puis du type du résultat. Par exemple '`fun [I I] S`' est une fonction qui prend deux entiers en arguments, et retourne une chaîne de caractères.

L'expression *TypeMono* définit les types monomorphes (non polymorphes) : la seule différence avec *Type* est l'absence des variables liées **un**.

2.2 Sources

```

Metal      = Definition*
Definition = fun Function Args = Program ;;
           | var Var( = Val) ;;
           | proto Function Nbargs ;;
           | proto Function = Type ;;
           | type TypeName ;;
           | type TypeName = [ Fields ] ;;
           | type TypeName = TypeConstr ;;

```

<i>Program</i>	=	<i>Expr</i>		<i>Expr ; Program</i>	
<i>Expr</i>	=	<i>Arithm</i>		<i>Arithm :: Expr</i>	
<i>Arithm</i>	=	<i>A₁</i>		<i>A₁ && Arithm</i>	<i>A₁ Arithm</i>
<i>A₁</i>	=	<i>A₂</i>		! <i>A₁</i>	
<i>A₂</i>	=	<i>A₃</i>		<i>A₃ == A₃</i>	<i>A₃ != A₃</i>
		<i>A₃ < A₃</i>		<i>A₃ > A₃</i>	<i>A₃ <= A₃</i>
		<i>A₃ >= A₃</i>		<i>A₃ =. A₃</i>	<i>A₃ !=. A₃</i>
		<i>A₃ <. A₃</i>		<i>A₃ >. A₃</i>	<i>A₃ <=. A₃</i>
		<i>A₃ >=. A₃</i>			
<i>A₃</i>	=	<i>A₄</i>		<i>A₄ + A₃</i>	<i>A₄ - A₃</i>
		<i>A₄ +. A₃</i>		<i>A₄ -. A₃</i>	
<i>A₄</i>	=	<i>A₅</i>		<i>A₅ * A₄</i>	<i>A₅ / A₄</i>

A_5	=	$A_5 \% A_4$		$A_5 * . A_4$		$A_5 /. A_4$
		A_6		$A_6 \& A_5$		$A_6 A_5$
A_6	=	$A_6 \wedge A_5$		$A_6 \ll A_5$		$A_6 \gg A_5$
		<i>Term</i>		$\sim A_6$		$\sim A_6$
		$\sim . A_6$		- int		- float
		float				
<i>Term</i>	=	(<i>Program</i>)				
		int		'char'		nil
		string		<i>Xml</i>		
		[<i>NameOfField</i> : <i>Expr</i> (<i>NameOfField</i> : <i>Expr</i>)*]				
		[<i>Expr</i> *]		{ <i>Expr</i> * }		
		<i>Var</i> (. <i>Term</i>)*		set <i>Var</i> (. <i>Term</i>)* = <i>Expr</i>		
		<i>Var</i> (. <i>NameOfField</i>)*		set <i>Var</i> (. <i>NameOfField</i>)* = <i>Expr</i>		
		<i>Function</i> <i>Args</i> _{<i>Function</i>}		# <i>Function</i>		
		#{ <i>Expr Expr Type</i> }				
		let <i>Expr</i> -> <i>Locals</i> in <i>Expr</i>				
		if <i>Expr</i> then <i>Expr</i> else <i>Expr</i>				
		while <i>Expr</i> do <i>Expr</i>				
		for <i>Local</i> = <i>Expr</i> ; <i>Expr</i> ; <i>Expr</i> do <i>Expr</i>				
		for <i>Local</i> = <i>Expr</i> ; <i>Expr</i> do <i>Expr</i>				
		call <i>Expr Expr</i>				
		update <i>Expr</i> with [{ $_ , Expr$ }*]				
		<i>Constr Expr</i>		<i>Constr0</i>		match <i>Expr</i> with <i>Case</i>
<i>Args</i> _{<i>F</i>}	=	<i>Expr ...Expr</i> : autant de fois <i>Expr</i> que la fonction <i>F</i> a d'arguments				
<i>Args</i>	=	nothing		<i>Local Args</i>		
<i>Locals</i>	=	<i>Local</i>		(<i>Locals</i> ':: <i>Locals</i>)		
<i>Locals</i> '	=	<i>Local</i>		[<i>Locals</i> '']		
<i>Locals</i> ''	=	{ $_ , Locals$ }*				
<i>Val</i>	=	<i>Val</i> ₃		<i>Val</i> ₃ :: <i>Val</i>		
<i>Val</i> ₃	=	<i>Val</i> ₄		<i>Val</i> ₄ + <i>Val</i> ₃		<i>Val</i> ₄ - <i>Val</i> ₃
		<i>Val</i> ₄ + . <i>Val</i> ₃		<i>Val</i> ₄ - . <i>Val</i> ₃		
<i>Val</i> ₄	=	<i>Val</i> ₅		<i>Val</i> ₅ * <i>Val</i> ₄		<i>Val</i> ₅ / <i>Val</i> ₄
		<i>Val</i> ₅ % <i>Val</i> ₄		<i>Val</i> ₅ * . <i>Val</i> ₄		<i>Val</i> ₅ / . <i>Val</i> ₄
<i>Val</i> ₅	=	<i>Val</i> ₆		<i>Val</i> ₆ & <i>Val</i> ₅		<i>Val</i> ₆ <i>Val</i> ₅
		<i>Val</i> ₆ ^ <i>Val</i> ₅		<i>Val</i> ₆ << <i>Val</i> ₅		<i>Val</i> ₆ >> <i>Val</i> ₅
<i>Val</i> ₆	=	<i>Val</i> ₇		- <i>Val</i> ₆		~ <i>Val</i> ₆
		~ . <i>Val</i> ₆		- int		- float
		float				
<i>Val</i> ₇	=	int		'char'		nil
		string		<i>Xml</i>		[<i>Val</i> *]
		(<i>Val</i>)		{ <i>Val</i> * }		
		itof <i>Val</i>		ftoi <i>Val</i>		

Fields = *Field* | *Field, Fields*
Field = *NameOfField* | *NameOfField : TypeMono*

TypeConstr = *TypeConstr'* | *TypeConstr'* | *TypeConstr*
TypeConstr' = *Constr TypeMono* | *Constr0*

Case = *Case'* | *Case'* | *Case* | (*_ -> Program*)
Case' = (*Constr Local -> Program*) | (*Constr0 -> Program*)

Var = nom de variable
Function = nom de fonction
TypeName = *nom de type* | *nom de type(labels*)*

Local = variable locale (liée)
NameOfField = nom de champ dans une structure
Constr = constructeur de type
Constr0 = constructeur de type vide

Xml = < Tag (*Attribute**)>*Sub*</ Tag >
Sub = nothing
| Text
| *Xml Sub*
| Text *Xml Sub*
Attribute = label = string

int = entier
char = caractère
string = chaîne
float = flottant

Les entiers peuvent être codés dans les bases suivantes :

-en décimal : 12349
-en hexa : 0x3fe
-en binaire : 0b10011
-en octal : 0o234235

Ils sont codés sur 31 bits signés.

Les char permettent de récupérer le code ascii d'un caractère : 'A' est un entier qui vaut 65.

Les chaînes de caractères sont entre guillemets. Le caractère \ permet d'accéder à certaines commandes :

\n : retour chariot
\z : caractère NULL
\" : guillemet
\\ : \

\nombre en décimal : \132 est le caractère Ascii 132

Un `\` en fin de ligne permet de signaler au compilateur de ne pas tenir compte du retour à la ligne.

Les remarques sont, comme en C, entre `/*...*/` et peuvent être imbriquées les unes dans les autres.

3 Fondamentaux du langage

3.1 *Hello world*

On suppose l'existence d'une fonction **Secholn** de type 'fun [S] S', qui retourne l'argument, et qui, en effet de bord, affiche l'argument sur la sortie standard, suivi d'un retour à la ligne.

On suppose également qu'au démarrage, le système évalue la fonction **main** de type 'fun []I'.

Dans ce cas l'exemple 'Hello world' s'écrit simplement :

```
fun main=  
  Secholn "hello world" ;  
  0 ;;
```

Dans la suite on suppose l'existence des fonctions suivantes :

- **Secho** de type 'fun [S] S', équivalente à **Secholn**, mais sans le retour à la ligne
- **Iecholn** de type 'fun [I] I', équivalente à **Secholn**, mais pour les entiers
- **Iecho** de type 'fun [I] I', équivalente à **Secho**, mais pour les entiers

3.2 *Calcul et variables*

On peut définir une variable globale entière *x* initialisée avec la valeur '1' de la manière suivante :

```
var x=1;;
```

Dans l'exemple suivant, on veut calculer $x+y$, et $(x+y)^2$, en utilisant le premier résultat pour calculer le second, ce qui nécessite de créer une variable locale contenant $x+y$.

```
var x=123 ;;  
var y=456 ;;  
fun main=  
  let x+y->z in  
  (  
    Secho "x+y=" ; Iecholn z ;  
    Secho "(x+y)^2=" ; Iecholn z*z  
  ) ;  
  0 ;;
```

L'opérateur `let ... -> ... in ...` permet de créer une variable locale dont le scope est uniquement l'expression qui suit le « in ».

On peut modifier une variable globale grâce à l'opérateur `set ...=...` qui retourne la valeur passée en argument et qui, par effet de bord, remplace la valeur de la variable.

Dans l'exemple suivant, on veut calculer $x+y$ et placer le résultat dans z .

```
var x=123 ;;
var y=456 ;;
var z ;;
fun main=
  set z=x+y ;
  0 ;;
```

On remarque qu'il n'est pas nécessaire d'initialiser une variable globale. Dans ce cas sa valeur initiale est 'nil'.

'nil' est une valeur que peuvent prendre toutes variables, quelque soit leur type, et qui est équivalent à « vide ».

L'opérateur `if...then...[else ...]` est une fonction qui, en fonction du résultat de l'expression « condition » calcule l'expression « then » ou l'expression « else ». Le résultat de cette expression est le résultat de la fonction « if ». On peut donc intégrer cet opérateur dans une expression arithmétique :

```
1+if x==2 then 3 else 4
```

3.3 *Itération*

Le langage propose un opérateur d'itération : `for ... ; ... [; ...] do ...`

Dans le cas d'une itération « simple », on peut écrire par exemple : `for i=0 ; i<20 do ...`

L'expression qui suit le 'do' est alors évaluée 20 fois avec la variable locale i , créée pour l'occasion et dont le scope se limite à l'expression qui suit le 'do'.

Si l'itération n'est pas de type '+1' (par exemple on veut '+2'), on utilise la forme :

```
for i=0 ; i<20 ; i+2 do ...
```

(on écrit $i+2$, et non pas $i=i+2$, le 'i=' étant implicite)

Le langage propose également un opérateur `while ... do ...`

Il n'y a toutefois pas de commande « break » ou « continue ».

3.4 *Gestion de listes*

Les langages fonctionnels se prêtent bien à gestion de listes.

La manipulation des listes s'appuie sur trois opérateurs :

- `... :: ...` (double deux-points) : constructeur de liste 'fun [u0 list u0] list u0'
- `hd` : récupération du premier élément 'fun [list u0] u0'
- `tl` : récupération de la liste privée de son premier élément 'fun [list u0]list u0'

La liste vide vaut 'nil'

On écrit la fonction 'dumpListI' qui affiche le contenu d'une liste d'entiers.

```

fun dumpList l =
  if l==nil then SechoLn "nil"
  else
  (
    Iecho hd l; Secho " :: ";
    dumpList tl l
  );;

```

On peut également écrire :

```

fun dumpList l0 =
  for l=l0;l!=nil;tl l do (Iecho hd l; Secho " :: ");
  SechoLn "nil";;

```

Pour concaténer deux listes quelconques :

```

fun conc p q = if p==nil then q else (hd p) ::conc tl p q ;;

```

3.5 Gestion des tuples

Un tuple est un ensemble de valeurs quelconques ; on le note entre crochets, par exemple :

```
[123 "abc"]
```

On crée un tuple implicitement par cette écriture.

On accède aux éléments d'un tuple par l'opérateur let :

```
let tuple->[a b] in ...
```

Par exemple, on peut utiliser les tuples pour définir des vecteurs à 2 dimensions :

```

fun tup2_add a b =
  let a->[xa ya] in
  let b->[xb yb] in
  [xa+xb ya+yb];;

```

On peut modifier une ou plusieurs valeurs d'un tuple par l'opérateur 'update' :

```

let [123 "abc"] -> t in
(
  update t with [456 "def"] ;
  update t with [_ "def"];
  t
);

```

On évitera toutefois cet usage. Si on doit effectuer des effets de bord sur les tuples, on préférera utiliser des structures (voir plus loin).

3.6 Gestion des tables

Une table est un ensemble de valeurs de même type ; on la note entre accolades, par exemple :

```
{1 2 3}
```

On peut créer une table de deux manières :

- en utilisant l'accolade
- en utilisant l'opérateur tabnew : fun[u0 I] tab u0

L'opérateur tabnew prend en argument la valeur d'initialisation des éléments de la table ainsi que la taille d'un table.

On accède à un élément de la table de la manière suivante :

t.i : i-ème élément de la table t

Comme en C, les éléments sont numérotés à partir de zéro.

Si la valeur 'i' est hors limite (négative ou supérieure à la taille du tableau), la valeur retournée est 'nil'.

On peut changer la valeur d'un élément de la table avec l'opérateur set :

set t.i = t.i + 1 ;

3.7 Gestion des structures

Une structure est une sorte de tuple dont les champs sont nommés, ce qui permet d'y accéder plus simplement.

On doit d'abord définir les champs de la structure :

```
type AAA=[nameAAA scoreAAA] ;;
```

On peut alors créer une structure en écrivant :

```
[nameAAA : "foobar" scoreAAA :123]
```

On pourra accéder aux champs de la manière suivante :

```
Secholn s.nameAAA ;
```

On peut changer la valeur d'un élément de la structure par l'opérateur set :

```
set s.scoreAAA= 1+s.scoreAAA ;
```

3.8 Gestion des types somme

Les types somme sont l'équivalent du 'union' du langage C. On peut l'utiliser pour implémenter les automates, ou certains arbres d'évaluation.

On définit par exemple les différents états des noeuds d'un arbre :

```
type MySum= Zero | Const _ | Add _ | Mul _ ;;
```

Le caractère 'underscore' indique qu'un paramètre est associé au type somme.

```
fun eval z=
  match z with
  ( Zero -> 0)
  ( Const a -> a)
  |( Add [x y] -> (eval x)+(eval y))
  |( Mul [x y] -> (eval x)*(eval y)) ;;
```

On crée alors l'arbre de l'expression $1+(2*3)$ de la manière suivante :

```
Add [ Const 1 Mul [ Const 2 Const 3]]
```

3.9 Manipulation de fonctions

Le langage permet la manipulation de fonctions ; pour obtenir une sorte de « pointeur » vers une fonction, on utilise l'opérateur #. On utilise alors l'opérateur « call » pour appeler une fonction à partir de son pointeur.

```
fun compare x y= x-y ;;
```

```
fun main =  
  compare 1 2 ;  
  let #compare -> f in  
  IechoIn call f [1 2] ;; //call : fun[ fun u0 u1 u0]u1
```

On peut fixer le dernier argument d'une fonction et obtenir ainsi une fonction avec un argument de moins.

```
fun main=  
  let #compare -> f in  
  let fixarg2 f 2 -> g in  
  IechoIn call g [1] ;;
```

Dans cet exemple, la fonction g est la fonction de comparaison avec l'entier '2'.
On utilise l'opérateur fixarg n , avec $n=1, 2, 3, \dots$

4 Exemples simples

4.1 Génération d'une liste d'entiers aléatoires

On suppose l'existence d'une fonction 'rand' qui retourne un nombre aléatoire sur 16 bits.

```
fun mkrandomlist n=  
  if n>0 then rand ::mkrandomlist n-1 ;;
```

4.2 Tri par insertion d'une liste d'entiers

```
fun insert x l=  
  if l==nil then x::nil  
  else if (x-hd l)>0 then (hd l)::insert x tl l  
  else x::l;;
```

```
fun sort l= if l!=nil then insert hd l sort tl l;;
```

