METAL project

Grammar

Author: Sylvain Huet

Created: 13/01/03
Last Updated: 20/02/07

1. Introduction

1. *Overview of the document*

language Grammar Metal.

2. Description

1. *Types*

Type = *B* | *B*(*labels**)
| **u***n* | **w***n*
| **r***n*
| **list***Type* | **tab***Type* | [*Type**]
| **fun***[Type*] Type*

TypeMono = *B* | *B*(*labels**)
| **w***n* | **r***n*
| **list***TypeMono* | **tab***TypeMono* | [*TypeMono**]
|**fun***[TypeMono*] TypeMono*

B	=	Base type
u*n*	=	bound variable
r*n*	=	recursion level *n*

The basic types are:

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

This list is not exhaustive due to the structures and type constructors, you can develop your own basic types.

Some comments on the board, if you are not familiar with these notations.

The first line defines the expression *type,*which is actually the type Scol. Then, separated by '|', there are different ways to write the expression: it could be:
- *B,*which is defined in the third line: this is a basic type, such as I, S, F ... So, since I is a basic type, *B* can be written I, and as *type* can be written *B,* type I is a Metal.

- **u***n* where *n* is an integer: u0, u1, u2, u3, .. . of a kind: they correspond to a bound variables.
- **w***n:* where *n* is an integer w0, w1, w2, w3, ... types are low.
- **r***n: n* is an integer r0, r1, r2, r3, ... are types: they define the recursion in types.
- **tab** *Type:* array type. The word **tab** is followed by the type of the array elements. For example `I tab` is the type of an array of integers.
- **list** *Type:* array type. The word **list** is followed by the type of items in the list. For example `I list` is the type of a list of integers.
- **[***Type****]:** tuple type. Brackets, we write several *types* (the meaning of the star *). For example, `[IS]` is a tuple of two elements: the first is an integer and the second string. Optionally, the tuple is empty: `[]` **.** The tuple may itself contain tuples: `[I [S I]]`
- **fun***[Type*] Type:* function type. The word *fun* is followed by a tuple containing the arguments of the function and the type of the result. For example '`fun [I I] S`' is a function that takes two integers as arguments and returns a string.

The term *TypeMono* defines the monomorphic types (non-polymorphic) with the only difference *type* is the absence of **u***n* variables.

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* **;;**

Program = *Expr* | *Expr* **;** *Program*
Expr = *Arithm* | *Arithm* **::** *Expr*
Arithm = A_1 | A_1 **& &** *Arithm* | A_1 | | *Arithm*
A_1 = A_2 | ! AA_1
A_2 = A_3 | A_3 == A_3 | A_3 **!** = A_3
| A_3 < A_3 | A_3 > A_3 | A_3 <= A_3
| A_3 > = A_3 | A_3 =**.** A_3 | A_3 **!** =**.** A_3

| $A_3 <. A_3$ | $A_3 >. A_3$ | $A_3 <=. A_3$

| $A_3 >=. A_3$

A_3 = A_4 | $A_4 + A_3$ | $A_4 - A_3$

| $A_4 +. A_3$ | $A_4 -. A_3$

A_4 = A_5 | $A_5 * A_4$ | A_5 / A_4

| $A_5 \% A_4$ | $A_5 *. A_4$ | $A_5 /. A_4$

A_5 = A_6 | $A_6 \& A_5$ | $A_6 | A_5$

| $A_6 \wedge A_5$ | $A_6 << A_5$ | $A_6 >> A_5$

A_6 = *Term* | $-A_6$ | $\sim A_6$

| $-. A_6$ | **-** int | **-** float

| float

Term = **(** *Program* **)**

| int | 'char' | **nil**

| string | *Xml*

| **[***NameOfField* **:** *Expr* (*NameOfField* **:** *Expr)** **]**

| **[***Expr** **]** | {*Expr** }

| *Var*(**.***Term) ** | **set** *Var*(**.***Term) * = Expr*

| *Var*(**.***NameOfField) ** | **set** *Var*(**.***NameOfField) * = Expr*

| *Function* $Args_{Function}$ | *#Function*

| *# Expr Expr Type*{}

| **Let** *Expr* **->** *Locals* **in** *Expr*

| **if** *Expr* **then** *Expr* **else** *Expr*

| **while** *Expr* **do** *Expr*

| **for** *Local* = *Expr* **,** *Expr, Expr* **do** *Expr*

| **for** *Local* = *Expr* **;** *Expr* **Do** *Expr*

| **call** *Expr Expr*

| **Update** *Expr* **with [** _ *{, Expr}* * **]**

| *Constr Expr* | *Constr0* | **match** *Expr* **with** *Hut*

$Args_F$ = *Expr ...:Expr* many times *Expr* the function F of arguments

Args = nothing | *Local args*

Locals = *Local* | (*Locals'***::***Locals)*

Locals' = *Local* | **[** *Locals"* **]**

Locals" = *_{,} * Locals*

Val = Val_3 | Val_3 **::** *Val*

Val_3 = Val_4 | $Val_4 + Val_3$ | $Val_4 - Val_3$

| $Val_4 +. Val_3$ | $Val_4 -. Val_3$

Val_4 = Val_5 | $Val_5 * Val_4$ | Val_5 / Val_4

| $Val_5 \% Val_4$ | $Val_5 *. Val_4$ | $Val_5 /. Val_4$

Val_5 = Val_6 | $Val_6 \& Val_5$ | $Val_6 | Val_5$

| $Val_6 \wedge Val_5$ | $Val_6 << Val_5$ | $Val_6 >> Val_5$

Val_6 = Val_7 | $-Val_6$ | $\sim Val_6$

| $-. Val_6$ | **-** Int | **-** float

```
            |       float
Val7        =       int            |    'char'           |    nil
            |       string         |    Xml              |    [ Val * ]
            |       (Val)          |    { Val* }
            |       itof Val       |    ftoi Val

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            = variable name
Function       = function name
TypeName    =      type name        |        type name( labels* )

Local          = local variable (related)
NameOfField    = Field name in a structure
Constr         = type constructor
Constr0        = empty type constructor

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

int          =       integer
char         =       character
string       =       string
float        =       floating
```

Integers can be encoded in the following basis:

- decimal	:	12349
- hexa	:	0x3fe
- binary	:	0b10011
- octal	:	0o234235

They are coded on signed 31 bits.

The char used to retrieve the ASCII code of a character: 'A is an integer that is 65.

The strings are quoted. The \ character can access some commands:

\n	:	newline
\z	:	NULL character
\"	:	quote
\\	:	\
\decimal nuber:		\132 is ASCII character 132

A \ at end of line to signal the compiler to ignore the newline.

The comments are, as in C, between /*...*/ and can be nested inside each other.

3. Language Fundamentals

1. *Hello world*

We assume the existence of a function **Secholn** type 'fun [S] S', which returns the argument, and which, side effect, shows the argument to standard output, followed by a return to line.

It is also assumed that at startup, the system evaluates the function **main** of type 'fun [] I'.

In this case the example 'Hello world' is written simply:

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

In the following we assume the existence of the following:

- **Secho** type 'fun [S] S', equivalent to **Secholn,** but without the newline
- **Iecholn** type 'fun [I] I', equivalent to **Secholn**, but for integers
- **IECHO** type 'fun [I] I', equivalent to **Secho,** but for the whole

2. *calculation and variables*

A global variable ***x*** can be defined with initial value '1 'in the following way:

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

In the following example, we want to compute $x + y$ and $(x + y)^2$, using the first result to calculate the second, which requires a local variable containing value of $x + y$.

```
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 ;;
```

The operator *let* ... -> ... *in* ... creates a local variable whose scope is only the expression that follows the "in".

You can modify a global variable using the operator set ...=... that returns the value passed as an argument and, as a side effect, replaces the value of the variable.

In the following example, we want to compute `x + y` and place the result in `z`.

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

Note that it is not necessary to initialize a global variable. In this case the initial value is 'nil'. 'Nil'is a value that can take all variables, regardless of type, which is equivalent to "empty".

The operator *if ... then ... [else ...]* is a function that, depending on the result of "status" calculates the term "then" or the word "else". The result of this expression is the result of the "if". We can integrate this operator in an arithmetic expression:

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

3. *Iteration*

The language provides an operator iteration: *for ... ; ... [, ...] do ...*
In case of an iteration "simple", we can write for example: *for i = 0; i <20 do ...*
The expression following the 'do' is then evaluated 20 times with the local variable i, created for the occasion and whose scope is limited to the expression following the 'do'.

If the iteration is of type '+1' (eg. we want '+2'), use the form:

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

(we write i+2, not i = i +2, 'i=' is implied)

The language also provides an operator while: while ... do ...
However, there is no command "break" or "continue".

4. *Lists*

Functional languages are well suited to managing lists.

Manipulation of lists based on three operators:

- ... *::* ... (double colon): creates a list 'fun [u0 list u0] list u0''
- *hd*: gets the first element 'fun [list u0] u0'
- *tl*: retrieving the list without its first element 'fun [list u0] list u0

The empty list is 'nil'

We write the function 'dumpListI' that displays the contents of a list of integers.

```
fun dumpListI l=
      if l==nil then Secholn "nil"
      else
      (
            Iecho hd l; Secho "::";
            dumpListI tl l
      );;
```

We can also write:

```
fun dumpListI l0=
      for l=l0;l!=nil;tl l do (Iecho hd l; Secho "::");
      Secholn "nil";;
```

To concatenate two lists of any kind:

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

5. *Tuples*

A tuple is a set of arbitrary values surrounded by brackets, eg.
[123 "abc"]

A tuple is created implicitly by the writing.
You can access the elements of a tuple by the operator let:

```
      let tuple-> [ab] in ...
```

For example, can be used to define tuples of vectors in two dimensions:

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

You can change one or more values of a tuple by the operator 'update':

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

should be avoided, however this use. Whether to perform side effects on the tuples, we prefer to use structures (see below).

6. *Table Type*

A table is a set of values of the same type, we note in braces, eg.

```
      {1 2 3}
```

You can create a table in two ways:

- by using the bracket
- using the operator `tabnew: fun [u0 I] u0 tab`

Tabnew operator takes as argument the initial value of the elements of the table and the size of a table.

You can access an item in the table as follows:
t.i : i-th element of the table t

As in C, the elements are numbered from zero.
If the 'i' is out of range (negative or greater than the array size), the return value is 'nil'.

You can change the value of an element of the table with the operator set:

```
      set t.i = t.i +1 ;
```

7. *Structure Type*

Structure is a kind of tuple whose fields are named, which can be accessed more easily.

One must first define the fields of the structure:
AAA = [nameAAA scoreAAA];

We can then create a structure by writing:

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

We can access the fields as follows:

```
    Secholn s.nameAAA;
```

You can change the value of an element of the structure set by the operator:

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

8. *Sum Types*

sum types are equivalent to the 'union' of the C language It can be used to implement automates or parse trees.

We define such different states of the nodes of a tree:

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

The character 'undescore' indicates that a parameter is associated with the sum type.

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

It then creates the tree of the expression 1 + (2 * 3) as follows:

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

9. *Functions*

The language allows the manipulation of functions for a kind of "pointer" to a function, using the operator #. We then use the operator "call" to call a function from its pointer.

```
compare xy = fun xy;;

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

We can set the last argument of a function and thus obtain a function with an argument less.

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

In this example, the function g is the function of comparison with the integer '2 '.
Fixarg We use the operator*n,*with *n = 1, 2, 3, ...*

4. Simple examples

1. *Generating a list of random integers*

We assume the existence of a function 'rand' that returns a random number of 16 bits.

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

2. *Insertion sort a list of integers*

```
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;;
```