Langages et traducteurs : définition et spécifications du langage

1. Définition

Notre langage sera à typage dynamique

1. Valeurs de type entier :
   x:=1; // assignation
   y:=2;
   z := x+y; // Opérations classiques
   a: = z % x;
   y=3; // ré-assignation

2. Valeurs de types string:
   x:="fifi ";
   y:="de lolo";
   z:=x+y; // z = "fifi de lolo";

3. Valeurs structurées : des enregistrements
   -> Permet d'avoir des "tableaux" avec ce qu'on veut dedans
   -> Permet de faire liste, structures de données, etc.

   ex : *tab := rec(0:2 1:1 2:10 3:-4 4:"Ceci est un tableau" "Ceci est un tableau":5);*
   x := tab.0 + tab.1;
   y := tab."Ceci est un tableau"; //Difficulté : différencier les int et string
   print(x); // Affiche 3

   structure de données : *dice := rec(1:4 2:1)* // Représente une paire de dés
   pers := rec(nom:"XXX" prenom:"YYY" date_naiss:"14/12/1989")
   On peut définir des sous-structures particulières de l'enregistrement. Par exemple, un tableau est un enregistrement dont tous les indices sont entiers, successifs et commencent à 0, et dont tous les types sont identiques (que des int, string, etc).

Structures de contrôle (Turing complet) :

```
x:=4;
y:=5;
if (x>y) then
{
        while(x>0)
        {
                x = x - 1;
```

```
        }
}
else
{
        while(y>0)
        {
                y = y-1;
        }
}

for(a:=0; a<5; a=a+1){
        println("ce message s'affiche 5 fois");
}
```

4. Fonctions :
   Nous pensons utiliser uniquement des fonctions qui renvoient une valeur ou 'null' si la fonction ne produit pas un output dont on a besoin.
   Que fait-on avec les void ? Est-ce qu'on renvoi null automatiquement ?
   ex : *function add(a1,a2)*
   ```
   {
       return a1 + a2; // Valeur à retourner
   }
   ...
   x = add(2,12);
   ```

5. Valeurs/variables locales : définir la portée entre accolades (comme en JAVA)
   ex :
   ```
   x:=1;
   ...
   {
       x:=2; // Erreur x déjà défini
   }
   ----------
   {
           x:=1;
   }
   ...
   {
           x:=2; // Ok portée correcte
   }
   ```

6. I/O : *print(x); // imprime 'x'*
   println(x); // imprime 'x' sur une ligne

read(); // lecture sur l'entrée standard

7. Choix paradigme : impératif (pour les changements d'états)

8. Choix supplémentaires : je suis partant pour (dans l'ordre de préférence) : mutli-threading, orienté objet, gestion fichiers, exceptions.

Concrete Grammar (BNF):

<letter> ::= <lowercase> | <uppercase>
<lowercase> ::= 'a' | 'b '| 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'y' | 'z'
<uppercase> ::= 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'
<ascii> ::= all the ascii characters ((0)10 to (127)10) with the restriction that the ascii character 34 (") must be preceded by the character 92 (\)
<string> ::= "" { <ascii> } ""
<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<number> ::= <digit> | <number>
<float> ::= <number> "." <number>
<boolean> ::= 'true' | 'false'
<value> ::= <number> | <string> | <boolean>
<variable> ::= <letter> | <digit> | <variable>
<entity> ::= <string> | <number> | <boolean> | <variable> | <record>
<record> ::= 'rec'({ (<expression>) ':' <expression> } ')'
<comparator> ::= '<' | '<=' | '==' | '!=' | '>=' | '>'
<logical operator> ::= '&&' | '||' | '!'
<arithmetic operator> ::= '-' | '+' | '/' | '*' | '%'
Nous considérons qu'il y a un ordre dans ces opérateur de sorte qu'on fait d'abord les multiplication/division/modulo avant les addition/soustraction.
<inc operator> ::= '++' | '--'
<inc expression> ::= <variable> <inc operator> <delimiter>
<separator> = <comparator> | <arithmetic operator>
<delimiter> ::= ;
<condition> ::= <boolean> | <expression> { (<comparator> | <logical operator>) <expression> }
<initialization> ::= <variable> ':=' <expression>
<affectation> ::= <variable> ':=' <expression>
<expression> ::= <inc expression> | <expression> { <separator> <expression> } | <string> { (<comparator> | '+') <string> } | <if expression structure> | <function structure> | <number> | <string> | <boolean> | <variable> | <record>
<instruction> ::= <expression> | <affectation> | <if instruction structure> | <loop>

<if expression structure> ::= <if expression> | <if else expression>
<if instruction structure> ::= <if instruction> | <if else instruction>
<if expression> ::= ‘if’ ‘(’ <condition> ‘)’ ‘then’ ‘{‘ { <expression> <delimiter> } ‘}’
<if else expression> ::= ‘if’ ‘(’ <condition> ’)’ ‘then’ ‘{‘ { <expression> <delimiter> } ‘}’ else’ ‘{‘ { <expression> <delimiter> } ‘}’
<if instruction> ::= ‘if’ ‘(’<condition> ‘)’ ‘then’ ‘{‘ { <instruction> <delimiter> } ‘}’
<if else instruction> ::= ‘if’ ‘(’<condition>’)’ ‘then’ ‘{‘ {<instruction> <delimiter> } ‘}’ else’ ‘{‘ { <instruction> <delimiter> } ‘}’
<loop> ::= <while structure> | <for structure>
<while structure> ::= ‘while’ ‘(’ <condition> ‘)’ ‘{’ { <instruction> <delimiter> } ’}’
<for structure> ::= ‘for’ ‘(‘ <affectation> ‘;’ <condition> ‘;’ <variable> <arithmetic operator> <number> ’)’ ‘{’ { <instruction> <delimiter> } ‘}’
<function structure> ::= ‘function’ <string> ‘(’ [[{ <variable> ‘,’}] <variable>] ’)’ ’{’ { <instruction> <delimiter> } ’}’
<function call> ::= <string> ‘(’ [[{ (<value> | <variable>) ‘,’}] (<value> | <variable>)] ’)’
<thread> ::= thread <variable> “{” { <instructions> <delimiter> } “}”

Comments:

{<x>} means that <x> can be repeated as much as needed.
When a character is surrounded with ‘, that means that it is the character itself.
[<x>] means that <x> is optional

Semantic:

Booleans = {true, false}
Entiers = $\mathbb{Z}$
Strings = {a* | a ∈ Chars}
Chars = {all the ascii characters ((0)10 to (127)10)}
Records = {label(x1:a1, x2:a2, …, xn:an) | label ∈ Strings, xi ∈ Strings (0<i<=n), ai ∈ Strings ∨ ai ∈ Booleans ∨ ai ∈ Records (0<i<=n)}
Functions = { ‘function’ label(x1, x2, …, xn) ‘{’ Instructions ‘}’ | label ∈ Srings}
FunctionCalls= { label(a1, a2, …, an) | label ∈ Srings}
Construct(lab(x1:y1 x2:y2 … xn:yn), E) = eval(lab, E) <eval(x1,E):eval(y1, E) eval(x2,E):eval(y2, E) ... eval(xn,E):eval(yn, E)>

eval(*0*) = 0
eval(*1*) = 1
.
.
.
eval(*n*) = n if n ∈ $\mathbb{Z}$

n if n ∈ Chars
n1.eval(n') if n ∈ Strings

Where *n1* is the first character of the string *n*
n' is the string *n* without the first character *n1*
a.b represents the concatenation of a character (*a*) and a string/character (*b*)

eval(*true*) = true
eval(*false*) = false

eval(*a* + *b,* E) = eval(*a*) + eval(b) if eval(*a*) ∈ ℤ ∧ eval(*b*) ∈ ℤ
eval(*a*) . eval(b) if a ∈ Strings ∧ b ∈ Strings
false otherwise
eval(*a* - *b,* E) = eval(*a*) - eval(*b*) if eval(*a*) ∈ ℤ ∧ eval(*b*) ∈ ℤ
false otherwise
eval(*a* / *b,* E) = eval(*a*) / eval(*b*) if eval(*a*) ∈ ℤ ∧ eval(*b*) ∈ ℤ\{0}
false otherwise
eval(*a* * *b,* E) = eval(*a*) * eval(*b*) if eval(*a*) ∈ ℤ ∧ eval(*b*) ∈ ℤ
false otherwise
eval(*a* % *b,* E) = eval(*a*) % eval(*b*) if eval(*a*) ∈ ℤ ∧ eval(*b*) ∈ ℤ
false otherwise
eval(*a* < *b,* E) = true if eval(*a*) ∈ ℤ ∧ eval(*b*) ∈ ℤ ∧ eval(*a*) < eval(*b*)
= true if eval(*a*) ∈ Strings ∧ eval(*b*) ∈ Strings ∧ Length(eval(*a*)) < Length(eval(*b*))
= false otherwise
eval(*a* <= *b,* E) = true if eval(*a*) ∈ ℤ ∧ eval(*b*) ∈ ℤ ∧ (eval(*a*) < eval(*b*) ∨ eval(*a*) = eval(*b*))
= true if eval(*a*) ∈ Strings ∧ eval(*b*) ∈ Strings ∧ (Length(eval(*a*)) < Length(eval(*b*)) ∨ Length(eval(*a*)) = Length(eval(*b*)))
= false otherwise
eval(*a* == *b,* E) = true if eval(*a*) ∈ ℤ ∧ eval(*b*) ∈ ℤ ∧ eval(*a*) = eval(*b*)
= true if eval(*a*) ∈ Strings ∧ eval(*b*) ∈ Strings ∧ Length(eval(*a*)) = Length(eval(*b*))
:= false otherwise
eval(*a* > *b,* E) = true if eval(*a*) ∈ ℤ ∧ eval(*b*) ∈ ℤ ∧ eval(*a*) > eval(*b*)
= true if eval(*a*) ∈ Strings ∧ eval(*b*) ∈ Strings ∧ Length(eval(*a*)) > Length(eval(*b*))
= false otherwise
eval(*a* >= *b,* E) = true if eval(*a*) ∈ ℤ ∧ eval(*b*) ∈ ℤ ∧ (eval(*a*) > eval(*b*) ∨ eval(*a*) = eval(*b*))
= true if eval(*a*) ∈ Strings ∧ eval(*b*) ∈ Strings ∧ (Length(eval(*a*)) > Length(eval(*b*)) ∨ Length(eval(*a*)) = Length(eval(*b*)))
= false otherwise

Where *Length(a)* corresponds to the number of characters that the string *a* contains.

eval(*a && b,* E) = true if evalBool(a) = true ∧ evalBool(*a*) = true
= false otherwise
eval(*a* || *b,* E) = true if evalBool(a) = true ∨ evalBool(a) = true
= false otherwise
eval(! a, E) = true if evalBool(a) = false
= false if evalBool(a) = true

Where evalBool(a, E) = false if eval(a, E) = 0

= true if eval(a, E) != 0

instr(*x:=a*, E) = E' = E U {x -> eval(a, E)} if a ∈ ℤ ∨ a ∈ Strings ∨ a ∈ Booleans, a ∈ Functions ∨ a ∈ Records
instr(*x=a*, E) = E' = E ⊕{x -> eval(a, E)} if a ∈ ℤ ∨ a ∈ Strings ∨ a ∈ Booleans, a ∈ Functions ∨ a ∈ Records

instr(function a(x1, x2, …, xn) { Instructions }, E) = E U {a -> } // demander au seb
eval(*a(a1, a2, …, an)*, E) = eval(Instructions, E') if a(a1, a2, …, an) ∈ FunctionCalls
Where E' = E ⊕{x1=a1, x2=a2, …, xn=an}
Instructions represents all the instructions from the body of the function *a*

eval(*struct*, E) = σ = σ ⊕{ ζ -> struct} ∧ E'=E ∧ σ if struct ∈ Records
eval(*struct.x*, { struct -> ζ }) = σ(ζ)[x] if struct ∈ Records
eval(*struct.x=a*, E) := σ(ζ)[x] = a avec ζ = E(struct) = eval(struct, E) ∧ E'=E
Where σ is a function that links references to records present in the store

instr(*if* (Condition) *then* {Instructions}, E) = instr(Instructions, E) if eval(Condition, E) = true
instr(*if* (Condition) *then* {Instructions_t} else {Instructions_f}, E) :=
instr(Instructions_t, E) if eval(Condition, E) = true
instr(Instructions_f, E) if eval(Condition, E) = false

instr(while (Condition) {Instructions}, E) = E' = instr(Instructions, E) while eval(Condition, E') = true
instr(for (Affectation, Condition, Instruction) {Instructions_body}, E) =
E' = instr(Affectation, E)
∧
instr(E'' = Instructions_body, E') ∧ instr(E''' = Instructions, E'') while eval(Condition, E'''')
= true
instr(thread a {Instructions}, E) = instr(Instructions, E)