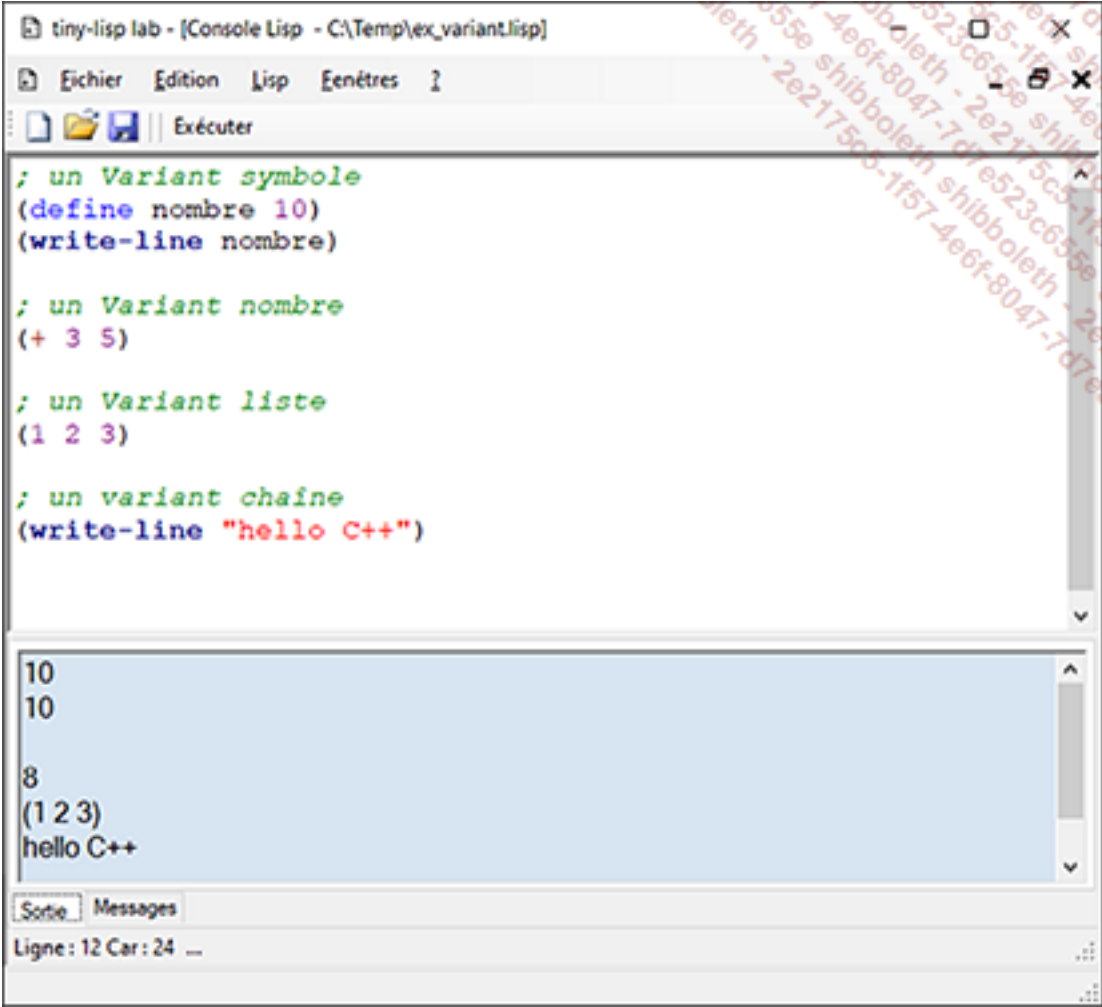


# Travaux pratiques

L'interprète tiny-lisp s'appuie largement sur la STL. Voici des précisions sur la classe `Variant` implémentée à grand renfort d'objets issus de la bibliothèque standard.

## 1. La classe Variant

`Variant` est le type de données universel de tiny-lisp. Cela peut être un symbole, un nombre, une liste (de `Variant`), une procédure.



```
tiny-lisp lab - [Console Lisp - C:\Temp\ex_variant.lisp]
Fichier  Edition  Lisp  Fenêtres  ?
Exécuter

; un Variant symbole
(define nombre 10)
(write-line nombre)

; un Variant nombre
(+ 3 5)

; un Variant liste
(1 2 3)

; un variant chaine
(write-line "hello C++")

10
10

8
(1 2 3)
hello C++

Sortie  Messages
Ligne : 12 Car : 24
```

Dans tiny-lisp, l'objet `Variant` fait partie d'un environnement, un conteneur doté d'une table des symboles. Cette structure est nécessaire à l'exécution des fonctions et des

lambda-expressions LISP pour passer les paramètres et créer des variables locales.

```
enum variant_type
{
    Symbol, Number, List, Proc, Lambda, Chaine
};

// definition à venir ; Variant et Environment se référencent mutuellement
struct Environment;

// un Variant représente tout type de valeur Lisp
class Variant {
public:

    // fonction qui renvoie Variant et qui prend comme argument variants

    typedef Variant(*proc_type) ( const std::vector<Variant>& );

    typedef std::vector<Variant>::const_iterator iter;

    typedef std::map<std::string, Variant> map;

    // types pris dans l'énumération : symbol, number, list, proc ou lamda
    variant_type type;

    // valeur scalaire
    std::string val;

    // valeur list
    std::vector<Variant> list;

    // valeur lambda
    proc_type proc;

    // environnement
    Environment* env;

    // constructeurs
    Variant(variant_type type = Symbol) : type(type) , env(0), proc(0) {
```

```

    }

    Variant(variant_type type, const std::string& val) :
    type(type), val(val) , env(0) , proc(0) {

    }

    Variant(proc_type proc) : type(Proc), proc(proc) , env(0) {

    }

    std::string to_string();
    std::string to_json_string();
    static Variant from_json_string(std::string json);
    static Variant parse_json(jsonlib::Json job);
};

```

## 2. La méthode to\_string()

C'est une méthode récursive plutôt simple. Elle est rédigée dans un style STL assez direct sans chercher à substituer les types itérateurs par des **typedef**. Charge au lecteur d'améliorer la rédaction de ce programme :

```

std::string Variant::to_string()
{
    if (type == List)
    {
        std::string s("");
        for (Variant::iter e = list.begin(); e != list.end(); ++e)
        {
            Variant v = *e;
            s.append(v.to_string());
            s.append(" ");
        }

        if (s[s.size() - 1] == ' ')

```

```

        s.erase(s.size() - 1);

        return s + ")";
    }
    else if (type == Lambda)
        return "<Lambda>";

    else if (type == Proc)
        return "<Proc>";

    else if (type == Chaîne)
        return val;

    return val;
}

```

### 3. La traduction JSON

On utilise les termes sérialisation et désérialisation pour exprimer la traduction d'un format objet vers un autre format tel que XML ou JSON.

#### a. La méthode statique `to_json_string()`

Le code de sérialisation JSON est assez similaire à la méthode `to_string()` :

```

std::string Variant::to_json_string()
{
    const std::string cl = "{ \"cell\": ";

    if (type == List)
    {
        std::string s;
        s.append(cl);
        s.append("{ ");
    }
}

```

```

s.append("\"type\": \"list\", ");

s.append(" \"list\": [ ");

for (Variant::iter e = list.begin(); e != list.end(); ++e)
{
    Variant v = *e;
    s.append(v.to_json_string());
    s.append(",");
}

if (s[s.size() - 1] == ',')
s.erase(s.size() - 1);

s.append(" ] "); // list

s.append(" } ");

s.append(" ");
return s;
}

else if (type == Lambda)
{
    std::string clamb = cl;
    clamb.append("{ \"type\": \"lambda\" }");
    return clamb;
}

else if (type == Proc)
{
    std::string cproc = cl;
    cproc.append("{ \"type\": \"proc\" }");
    return cproc;
}

else if (type == Chaine)
{
    std::string sc = cl;
    sc.append("{ \"type\": \"string\", \"val\": ");
    sc.append("\"");
    sc.append(val);
    sc.append("\" }");
}

```

```

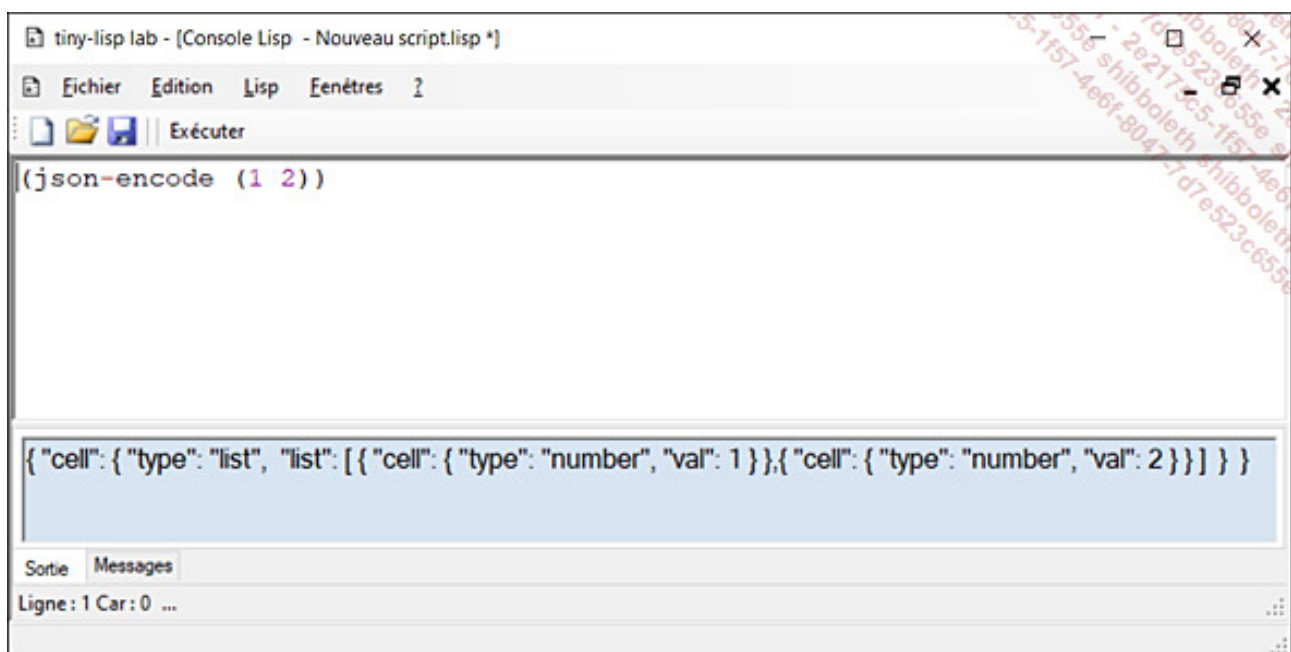
    return sc;
}

else if (type == Number)
{
    std::string sn = cl;
    sn.append("{ \"type\": \"number\", \"val\": ");
    sn.append(val);
    sn.append(" } }");
    return sn;
}

// symbol
std::string ss = cl;
ss.append("{ \"type\": \"symbol\", ");
ss.append(\" \"val\": \");
ss.append(val);
ss.append(\" } }");

return ss;
}

```



## b. La méthode statique from\_json\_string()

Pour la désérialisation JSON vers Variant, le code s'appuie sur une librairie JSON11 qui constitue des paires clé-valeur au fil de la lecture. L'algorithme repose sur une méthode auxiliaire `parse_json()` un petit peu plus difficile :

```
#pragma region parse_json
Variant Variant::parse_json(jsonlib::Json job)
{
    Variant v;

    if (job.is_object())
    {
        for (auto items = job.object_items().begin(); items
            != job.object_items().end(); items++)
        {
            if ((*items).first == "cell")
            {
                //std::cout << "cell ";
                Variant e;
                e = parse_json((*items).second); // type, val
                v.list.push_back(e);
            }
            else if ((*items).first == "type") {
                std::string v_type = (*items).second.is_string() ?
                (*items).second.string_value() : "";
                if (v_type == "number")
                    v.type = Number;

                if (v_type == "string")
                    v.type = Chaine;

                if (v_type == "list")
                    v.type = List;
            }
            else if ((*items).first == "val") {
                if ((*items).second.is_string())
                    v.val = (*items).second.string_value();

                else if ((*items).second.is_number())
                    v.val =
                Utils::str((*items).second.number_value());
            }
        }
    }
}
```

```

    }
    else if ((*items).first == "list") {
        if ((*items).second.is_array()) {
            v.type = List;
            for (auto listitem =
(*items).second.array_items().begin(); listitem !=
(*items).second.array_items().end(); listitem++) {
                Variant le;
                le = parse_json(*listitem);
                v.list.push_back(le.list[0]);
            }
        }
    }
}
return v;
}
if (job.is_string())
{
    v.val = job.string_value();
}
if (job.is_number())
{
    v.val = Utils::str(job.number_value());
}
if (job.is_array())
{
    //déjà traité
}

return v;
}
#pragma endregion

#pragma region from_json_string
Variant Variant::from_json_string(std::string sjson)
{
    Variant s;
    std::string err;
    const auto job = jsonlib::Json::parse(sjson, err);
    s = parse_json(job);
}

```



```
return s.list[0];  
}  
#pragma endregion
```

