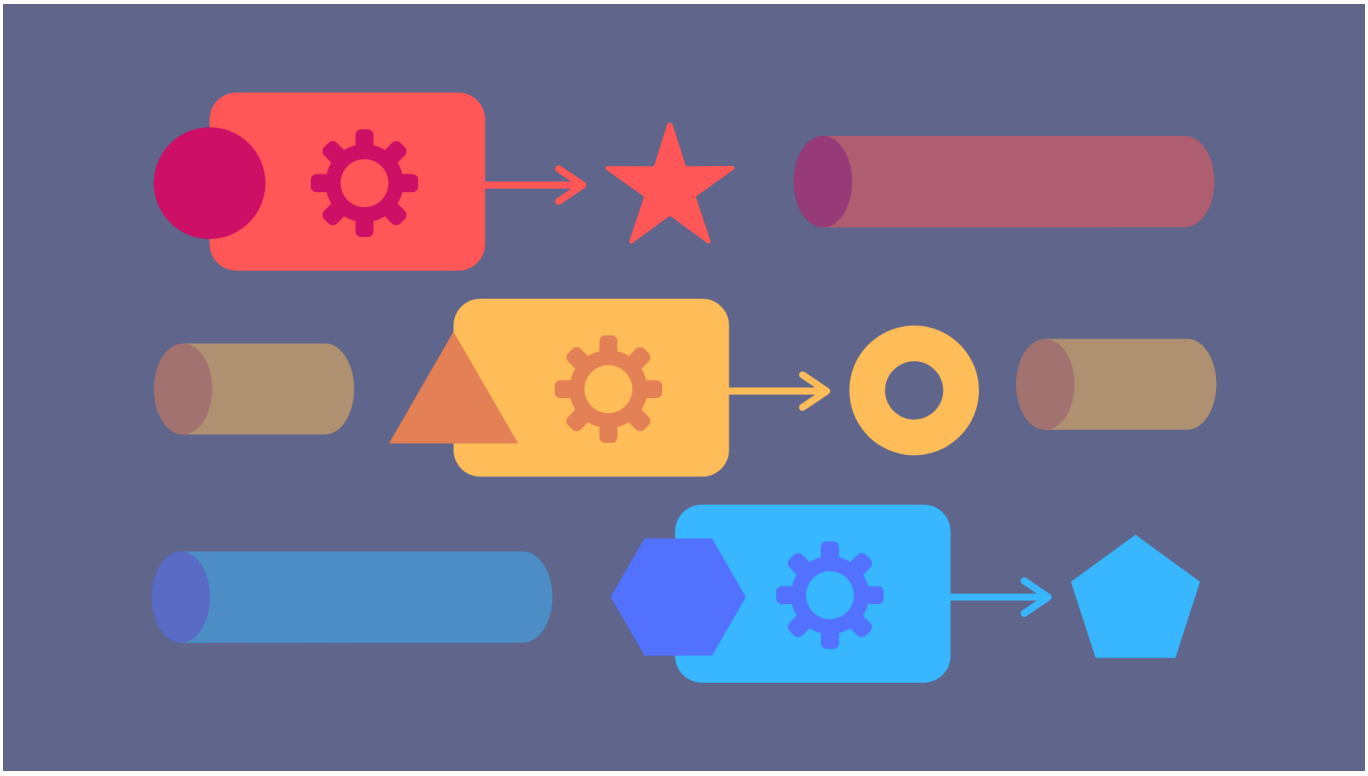


Toutes les illustrations sont de l'auteur.

À la découverte des parsers

Un parser est un traducteur. Un programme capable de prendre une information exprimée dans un format A, de la lire, de l'analyser, parfois même de l'enrichir, puis de rendre le tout dans un format B.

Pensez aux langages de programmation, formats de fichiers, protocoles de communication. Tous dépendent d'une multitude de parsers successifs pour fonctionner, que ce soit pour la compilation, l'interprétation, l'analyse syntaxique, le formatage automatique, la configuration, la correction orthographique... Nous utilisons constamment des programmes qui lisent du texte, identifient des motifs et en construisent des représentations internes qui peuvent être analysées.

L'objectif

Nous réaliserons un parser capable d'identifier une phrase (de façon simplifiée) et de renvoyer tout ses mots.

Nous utiliserons la technique de "recursive descent" pour construire notre parser final, ce qui mettra en lumière quelques bonnes idées de la programmation orientée fonction, un paradigme complémentaire à l'orienté objet et que beaucoup de programmeurs ne connaissent malheureusement pas.

Un outil de choix : Le CoddiScript

Nous utiliserons le CoddiScript. Un langage fictif conçu spécialement pour vous donner une vision haut-niveau de l'implémentation de notre parser.

CoddiScript est faiblement typé (mais on peut documenter les types) et accepte que les variables contiennent des fonctions (on dit que le langage a des "first-class functions").

Il est inspiré de Javascript et de Python, mais tout ce que je vais vous montrer peut être implémenté dans d'autres langages plus bas niveau et/ou avec des propriétés différentes. Voici un exemple en Rust et un autre en Java.

Petit exemple de CoddiScript pour vous familiariser :

```
// On peut créer des fonctions et documenter les typess
// @typespec f(int) -> List[int]
fun ma_fonction(n) {
    return [n, n * 10, n * 100]
}

println(ma_fonction(2)) // [2, 20, 200]

liste = ma_fonction(3)
println(liste[0]) // 3

// Les variables peuvent référencer des fonctions
f = ma_fonction
f(4) // [4, 40, 400]
```

Implémentation

0. Formule d'une phrase

Pour parser une phrase, il faut déjà identifier ce dont elle est composée. Dans notre cas ce sera d'**un ou plusieurs mots**, mots composés d'**un ou plusieurs caractères alphanumériques**, le premier commençant par **une majuscule**, le dernier finissant par un **point**, tous séparés d'**un espace**.

Nous allons nous limiter à ces quelques règles pour simplifier. Mais si vous souhaitez ajouter des règles de grammaire et de syntaxe avancées (par exemple : sujet, verbe, complément) je vous invite à vous renseigner sur les "parse tree", un outil complémentaire que je ne traiterai pas dans cet article.

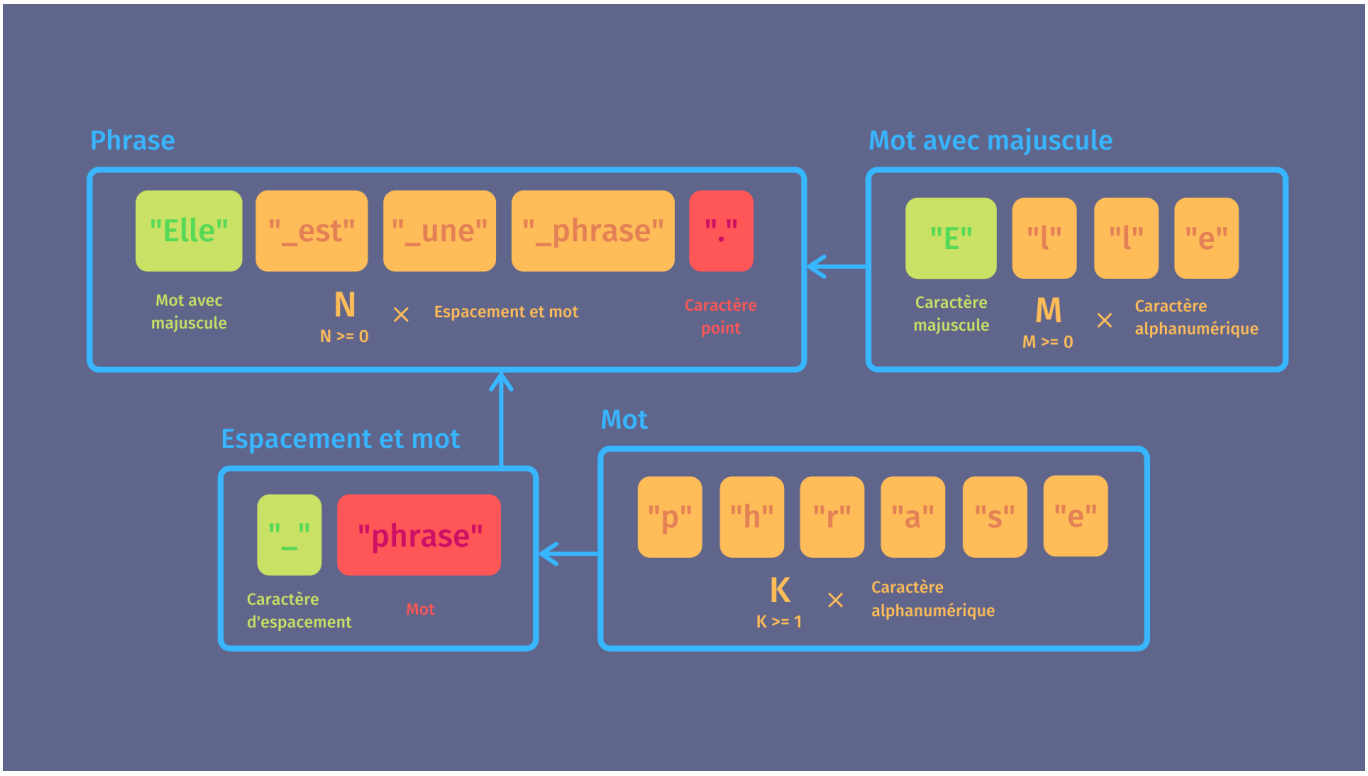
Exemple de phrases valides selon nos critères :

```
Je suis une phrase.     -> valide, resultat: ["Je", "suis", "une", "phrase"]
Je.                     -> valide, resultat: ["Je"]
je.                     -> incorrect
Je                      -> incorrect
Je suis une phrase .    -> incorrect
Je   suis une  phrase.  -> incorrect
```

On en déduit la formule pseudo-mathématique d'une "phrase" :

```
mot = n*char_alphanum
mot_maj = char_alphanum_maj + k*char_alphanum
```

```
phrase = mot_maj + k*(char_espacement + mot) + "."

n entier >= 1, k entier >= 0
```



Ce qui veut dire qu'on doit pouvoir parser :

- un caractère `"."`
- un caractère alphanumérique (`a`, `b`, `c` ... `8`, `9`, `0`)
- un caractère alphanumérique en majuscule
- un caractère considéré comme un espace (`" "`, `"\n"` ...)

Et faire des opérations sur ces parsers :

- répéter une ou plusieurs fois un parser (pour mot et espacement)
- répéter zero ou plusieurs fois un parser (pour les premiers mots d'une phrase)
- combiner deux parsers (pour tous les `+`)

1. Fonction parser

Nous prenons une approche orientée fonction, donc chaque parser sera une fonction. Ces fonctions renverront le résultat du parsing appliqué au texte donné en entrée.

```
res = parse_phrase("Ma phrase.")
println(res) // ["Ma", "phrase"]
```

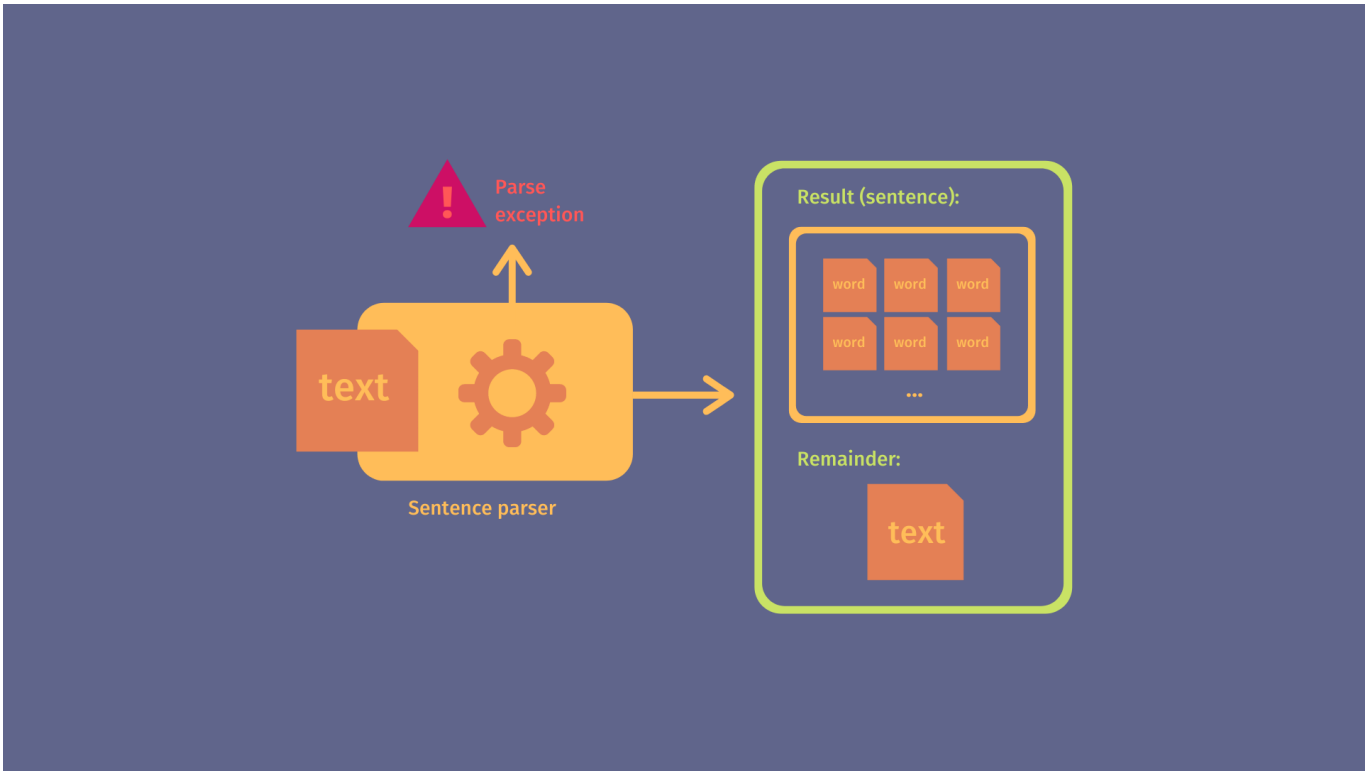
On voudra aussi renvoyer la partie pas traitée par le parser, pour pouvoir parser plusieurs fois à la suite.

```

```

```
(res, rem) = parse_phrase("Ma phrase1.Ma phrase2.suite")
println(res, rem) // (["Ma", "phrase1"], "Ma phrase2.suite")

(res, rem) = parse_phrase(rem)
println(res, rem) // (["Ma", "phrase2"], "suite")
```

Dans le cas où l'entrée ne sera pas valide on renverra une exception.

```
parse_phrase("  je ne suis pas une phrase")
// ParseException{
//      reason: "unexpected whitespace",
//      input: "  je ne suis pas une phrase"
// }
```



2. Parsers de base

Commençons par parser un unique caractère en le renvoyant en résultat :

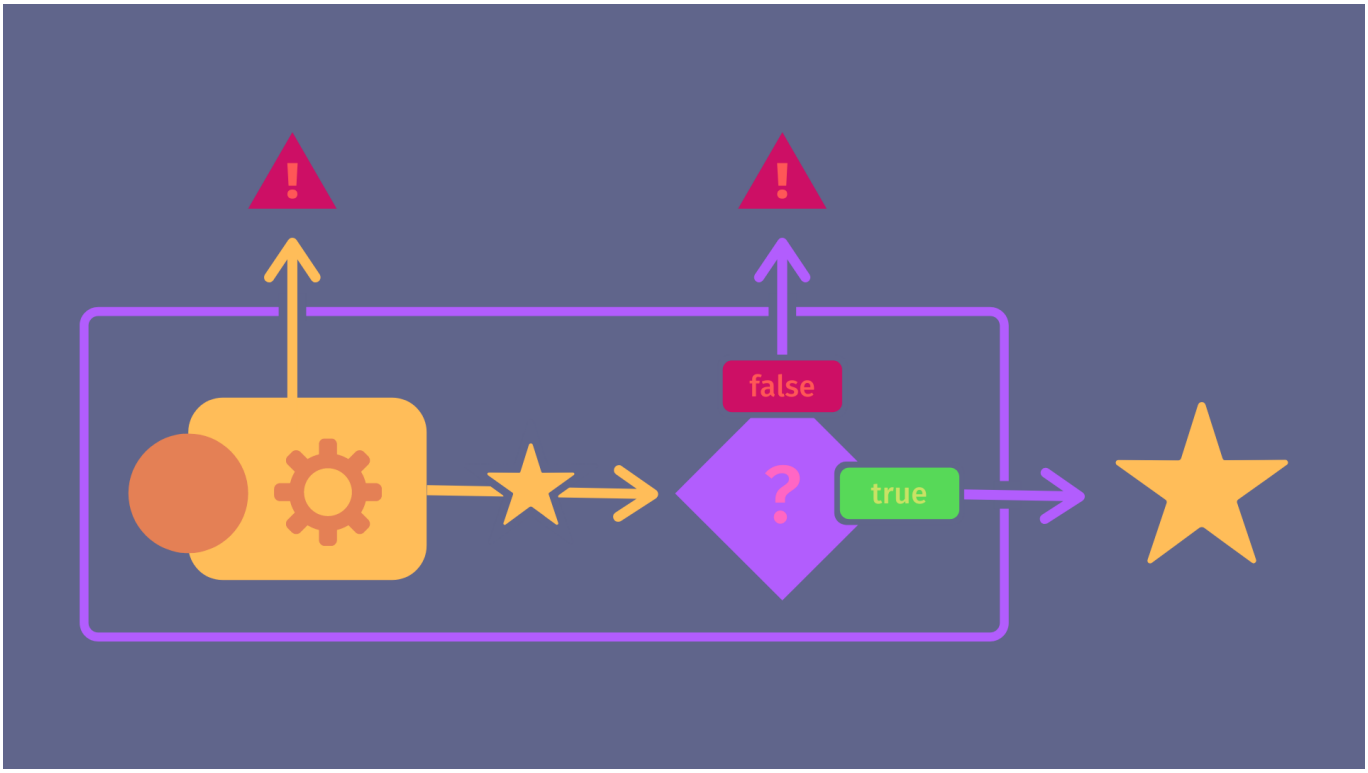
```
// Parse un caractère, renvoie le caractère et le reste de l'input
// @typespec f(str) -> (char, str)
fun parse_char(input) {
    if input.length < 1 { // pas de caractère du tout
        throw ParseException(input, "Expected a character, got nothing.")
    }
    return (input[1], input.substr(1))
    // input.substr(1) renvoie input sans son premier caractère
}
```

```
(res, rem) = parse_char("Hello")
println(res, rem) // ("H", "ello")

(res, rem) = parse_char(rem)
println(res, rem) // ("e", "llo")
```

3. Fonctions du premier ordre

Maintenant qu'on peut parser la présence d'un caractère, il faut vérifier certaines conditions dessus, par exemple si l'on veut parser un espace alors on doit regarder si c'est un espace, si l'on veut parser un point, si c'est un point et ainsi de suite...

Notre premier instinct serait de faire une fonction qui parse un caractère puis essaye de valider une condition booléenne dessus. Cette condition serait une fonction prenant le caractère en question et renvoyant vrai ou faux. Si la condition passe on renverra le caractère, sinon on aura une exception.




```
// Parser vérifiant la présence d'un caractère validant une condition
// @typespec f(str, (char) -> bool) -> (char, str)
fun parse_char_cond(input, cond) {
    (res, rem) = parse_char(input)
    if !cond(res) {
        throw ParseException(res, "Unexpected character.")
    }
    return (res, rem)
}

// Condition pour vérifier qu'un caractère est bien un point
// @typespec f(char) -> bool
fun is_dot(char) {
```

```
    return char == "."
}

// Parser final du caractère point
// @typespec f(str) -> (char, str)
fun parse_dot(input) {
    return parse_char_cond(input, is_dot)
}

(res, rem) = parse_dot(".suite")
println(res, rem) // (".", "suite")

(res, rem) = parse_dot("suite")
// ParseException{
//      reason: "Unexpected character",
//      input: "s"
// }
```

Pour `parse_char_cond` on parle de "fonction du premier ordre", c'est à dire une fonction qui prend d'autres fonctions en paramètre. C'est très utile car ça offre une forme de polymorphisme. La fonction `parse_char_cond` peut être réutilisée avec pleins de conditions différentes pour enrichir le comportement du parser sans dupliquer de code.

4. Functors

Le code écrit jusqu'ici n'est pas si bien. Imaginez un instant que vous vouliez ensuite faire un parser qui valide une condition sur autre chose qu'un caractère. Par exemple si maintenant vous vouliez parser un mot et en plus valider que ce mot a une majuscule, alors vous devriez réécrire une logique très semblable mais en remplaçant `parse_char` par `parse_word` (supposons qu'elle existe).

On doit rajouter un argument qui sera le parser à utiliser et renommer la fonction `parse_char_cond` en `parse_cond` pour marquer le fait qu'on ne parsera plus nécessairement qu'un seul caractère.

```
// @typespec<T> f(str, (str) -> (T, str), (T) -> bool) -> (T, str)
fun parse_cond(input, parser, cond) {
    (res, rem) = parser(input)
    if !cond(res) {
        throw ParseException(res, "Cound.")
    }
    return (res, rem)
}
```

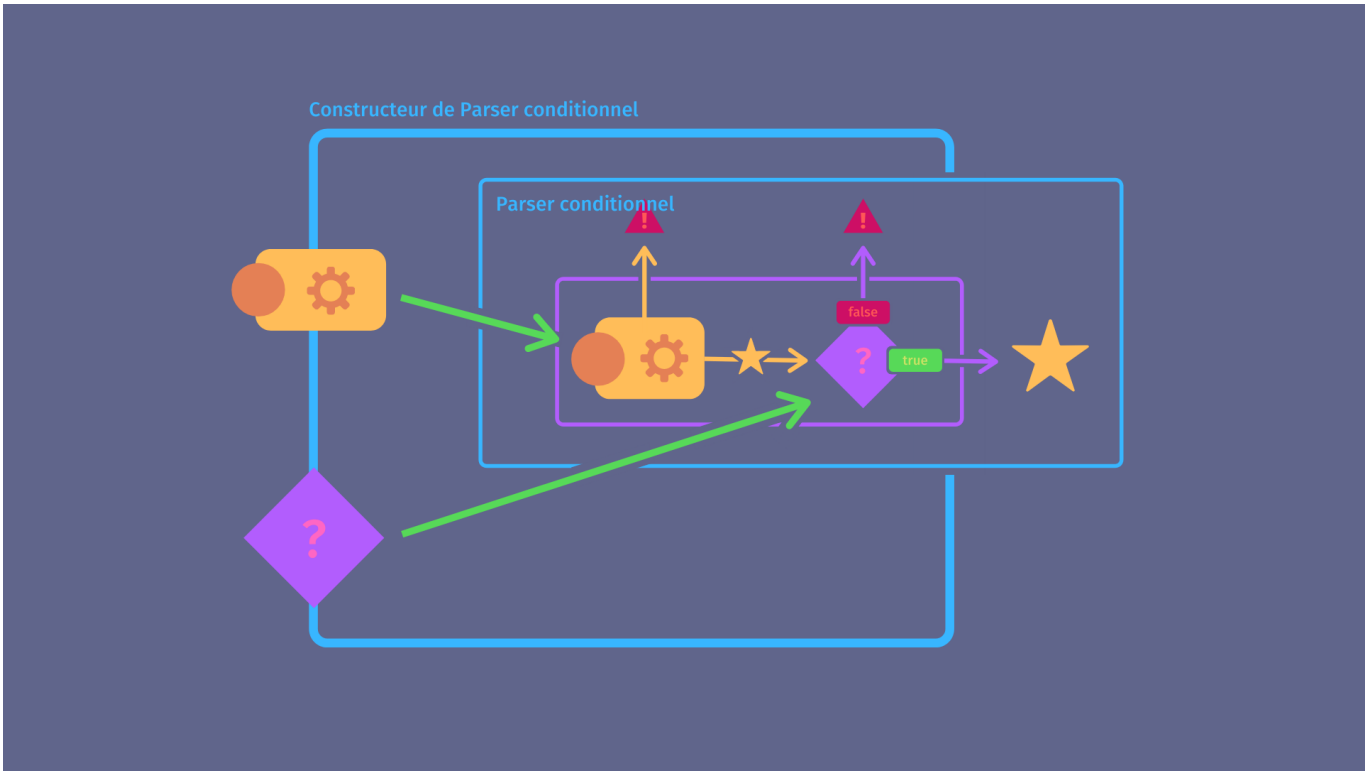
Mais on peut faire encore mieux.

Plus tard, le parser de phrase va utiliser `parse_cond` de la même manière plusieurs fois. Par exemple pour chaque caractère de chaque mot il va appeler `parse_cond` avec le même parser `parse_char` et la même condition `is_alphanumeric`. Et même si le temps nécessaire à l'envoi d'une fonction dans une autre est négligeable (pointeur de fonction => entier positif ~= 4 octets), si l'on fait ça sur des textes de plusieurs

millions de mots, on va répéter des millions de fois cet appel (on souhaite éviter les feux de datacenters à Strasbourg).

Pour résoudre ce problème nous n'allons pas faire un parser, mais une fonction qui va construire des variantes de `parse_cond` qu'on pourra appeler ensuite sans repasser tous les arguments.

```
// Comme le montre le typespec, cond_parser renvoie une fonction.
// C'est une sorte d'usine à fonctions.
// @typespec<T> f((str) -> (T, str), (T) -> bool) -> f(str) -> (T, str)
fun cond_parser(parser, cond) {
    // Construit une variante de parse_cond à l'aide de parser et de cond
    fun parse_cond(input) {
        (res, rem) = parser(input)
        if !cond(res) {
            throw ParseException(res, "Cound.")
        }
        return (res, rem)
    }
    return parse_cond
}
```



Lorsqu'une fonction en renvoie une autre on parle de "functor". On peut voir ça comme l'équivalent d'une "factory" en orienté objet.

Donc maintenant on peut réécrire notre parser de points comme ceci :

```
fun is_dot(c) { return c == "." }

dot_parser = cond_parser(parse_char, is_dot)
```

```
dot_parser(".super") // (".", "super")
```

Et pour parser les autres types de caractères qui nous intéressent :

```
import { is_whitespace, is_alphanum, is_uppercase } from "std:chars";
// La plupart des langages ont des fonctions similaires dans leur librairie
standard.

space_parser = cond_parser(parse_char, is_whitespace)
alphanum_parser = cond_parser(parse_char, is_alphanum)

maj_alphanum_parser = cond_parser(alphanum_parser, is_uppercase)
```

Vous voyez qu'avec cette technique nos parsers sont devenus très facilement composables. Regardez comme on utilise `alphanum_parser` pour construire `maj_alphanum_parser`. Ce qui forme une chaîne de parsers `parse_char -> alphanum_parser -> maj_alphanum_parser` où on ajoute simplement une nouvelle condition à chaque étape. Donc tout se compose très naturellement.

5. Parsers répétés

Nous allons maintenant créer un functor qui, à partir d'un parser A, créé un parser B répétant A autant de fois que possible jusqu'à ce qu'il échoue, renvoyant alors la liste de tous les résultats accumulés de A. Ce qui nous donnera la possibilité de répéter un parser zéro fois ou plus. Par exemple pour parser les caractères après la majuscule, comme indiqué dans la formule de départ.