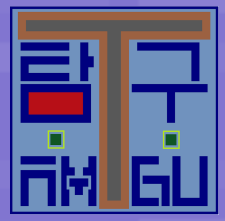
Présentation EPITA

Parsage...



Claude Roux

22 Mai 2024

Des théories en compétition

- ➢ Gouvernement et liage (*Chomsky*)
- ➢ *General Phrase Structure Grammar (Gazdar et Pullum 1985)*
- ➢ *Head-driven Phrase Structure*

GPSG présente deux difficultés:

a) Formalisme très complexe
b) Ordinateurs moins puissants que votre micro-onde.

J'ai fait mon doctorat sur un parseur pour GPSG à partir de 1990.

Formalisme

Le formalisme de GPSG est divisé en différents types de règles:

a) *DI-règles:* règle hors-contexte dont la partie droite est *un ensemble de catégories:*

$$SN \rightarrow \{nom, adj, det\}$$

b) PL-règles: règles de précédence qui indiquent l'ordre d'apparition des éléments.

$$[det] < [nom]$$
$$[det] < [adj]$$

La complexité du formalisme rend son informatisation au mieux comique.

Principe absolue

Pour qu'un programme aille vite, il doit exécuter le moins d'instructions possibles…

Implémentation

Comment implanter un algorithme efficace quand l'application d'une règle dépend non pas d'une séquence mais d'un ensemble…

En fait, la question est plutôt: *Comment peut-on gérer efficacement des ensembles sur un ordinateur???*

Avec des vecteurs?

L'intersection ou l'union de vecteurs, ça coûte à la machine du temps et de l'espace mémoire. L'intersection au mieux, c'est du $O(n^2)$.

De plus ça ne résout pas le problème de l'indexation.

Codage Binaire

Sauf que l'intersection et l'union de valeurs, l'ordinateur sait le faire de façon très efficace.

Si vos données sont codées sous la forme de bits.

A & B: effectue l'intersection des bits entre A et B
A | B: effectue l'union des bits entre A et B

Dans tous les cas de figure, ces opérations sont extraordinairement rapides.

Analyse Syntaxique

Avec des règles:

SN -> (Dét), (A),N.
SN -> Pro.
SV -> Pro*, V, (SN).
P -> SN, SV.

On peut analyser la phrase suivante avec cette grammaire:

La dame raconte une histoire

Codage binaire des Catégories

Catégorie	P	SN	SV	SA	SP
Puissance	0	1	2	3	4
Décimal	1	2	4	8	16
Binaire	0000000001	0000000010	0000000100	0000001000	0000010000
Catégorie	**V**	**N**	**A**	**Pro**	**Dét**
Puissance	5	6	7	8	9
Décimal	32	64	128	256	512
Binaire	0000100000	0001000000	0010000000	0100000000	1000000000

Codage d’une règle

Soit les règles suivantes:

- SN -> Det, A,N.
- SN -> Pro.
- SV -> Pro, V, SN.
- P -> SN, SV.

Chacune de ces règles peut-être ré-analysée comme étant une union de position binaire.

Par exemple, la règle: *SN -> Dét, A, N* correspond à:

512 | 128 | 64 = **1011000000 = 704**

Index

Règles	Décimal	Binaire
1.a) SN → N	64	0001000000
1.b) SN → N,Dét	576	1001000000
1.c) SN → N,A	192	0011000000
1.d) SN → N,A,Dét	704	1011000000
3) SV → Pro,V,SN	290	0100100010
5) P → SN,SV	6	0000000110

Ce code binaire a un double emploi:

a) Il indexe la règle
b) Il définit son contenu

576 c'est à la fois l'index de SN -> N,Dét, mais ça veut aussi dire une fois décomposée en valeurs binaires un nom et un déterminant…

Recherche d’une règle

Analysons: *le chat*

Soit donc la liste de catégorie: *Dét,N*

soit 512 | 64= 1000000000 | 0001000000 = 1001000000= 576

soit donc la règle: SN-> N, Dét.

Filtrage des règles

• L'utilisation d'un codage binaire permet aussi de prévoir rapidement si une règle peut ou non s'appliquer.
• En examinant notre tableau de règle, nous pouvons calculer à l'avance, les éléments suivants:

➢ Un *N* peut apparaître avec un *Dét* et un *A*
➢ Un *V* peut apparaître avec un *Pro*
➢ Un *SN* peut apparaître avec un *SV*

Nous allons traduire ces possibilités sous la forme de vecteurs binaires de filtrage.

Exemple de filtre

• Un filtre est donc composé des catégories qui peuvent être associées sous un même nœud syntagmatique.

Le filtre de *N* comprend aussi bien un *A* qu'un *Dét*.

•

•Nous pouvons traduire ce filtre sous une forme binaire.

Catégories	Filtre	Binaire
N	A,Dét,N	1011000000 = 704
A	A,Dét,N	1011000000 = 704
Dét	A,Dét,N	1011000000 = 704
V	Pro,V	0100100000 = 288
SN	SV,SN	0000000110 = 6
SV	SN,SV	0000000110 = 6

On peut même raffiné encore le processus, en contraignant ces filtres avec des règles de précédence.

Utilisation du filtre

Ces filtres sont utilisés lorsque l'on parcourt la phrase de la droite vers la gauche:

La dame lui lit une histoire

Dét^{1},N^{2}, Pro^{3}, V^{4}, Dét^{5}, N^{6}

A chaque étape on construit la liste des catégories sous la forme d'un vecteur de bits:

On rencontre d'abord N^{6} notre liste vaut: **0001000000**

Puis on rajoute Dét^{5}. On compare la liste avec le filtre de Dét:

0001000000 & *1011000000* = 0001000000

On retrouve le filtre, par conséquent Dét est compatible avec la liste.

Utilisation du filtre

La dame lui lit une histoire

Dét^{1},N^{2}, Pro^{3}, V^{4}, Dét^{5}, N^{6}

Peut-on rajouter le verbe V^{4}

1001000000 & 0100100000 = 00000000

Nous ne retrouvons pas notre filtre.
Le verbe n'est donc pas compatible avec notre liste.

Nous recherchons alors la règle qui correspond à cet index:

Sn -> Dét, N.

Ce qui nous donne donc SN.

Xerox Incremental Parser: XIP

XIP c’est:

a) Une dizaine de type de règles différentes
b) Des grammaires pour 8 langues dont le japonais.
c) Des compétitions où nous finissons toujours dans les 3 premiers
d) Des projets européens et français.
e) La grammaire de l’anglais comprend 60.000 règles et peut analyser des textes sur un PC de 2007 à une vitesse de 3000 mots/s.
f) Notre dernier vrai succès eut lieu en 2016 avec notre victoire à SemEval en Extraction de Sentiments.
g) La dernière compétition, organisée par IBM, eut lieu en 2017 et fut gagné par nous.

Type de règles

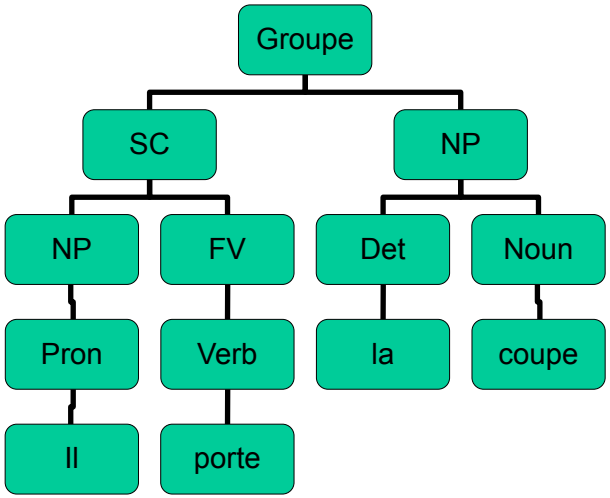
XIP offre différents types de règle:

a) Des règles de désambiguïsation. Le HMM *marque* les entrées qu'il considère comme bonne, avec un taux d'erreur d'environ 5%.
b) Des règles de *segments* pour construire un arbre syntaxique superficiel.
c) Des règles sémantiques, indexées sur des mots particuliers pour identifier des expressions figées.
d) Un moteur de prédicats du premier ordre pour identifier les dépendances syntaxiques.
e) Les règles sont organisées en couche de même type.
f) Des mécanismes d'encodage binaire à tous les niveaux permettent de n'appliquer qu'une infime partie des règles. Ainsi, pour une phrase donnée avec la grammaire de l'anglais comprenant 60.000 règles, seules une cinquantaine de règles seront tentées et peut-être seulement une vingtaine laisseront vraiment leur trace sur le résultat final.

Extraction des dépendances

Calcul des dépendances: *Il porte la coupe.*

|NP{?*,#1[last]}, VP{?*,#2[last]}| *Subj(#2,#1)*




```
SUBJ_PRON(porte,Il)
VARG_NOUN_DIR(porte,coupe)
DETERM_DEF_NOUN_DET(la,coupe)
CLOSEDNP_PRON(Il)
CLOSEDNP_DEF_DET(coupe)
```

0>GROUPE{SC{NP{Il} FV{porte}} NP{la coupe} .}

Les règles de dépendance

Il s'agit de règles qui manipulent les dépendances entre elles:

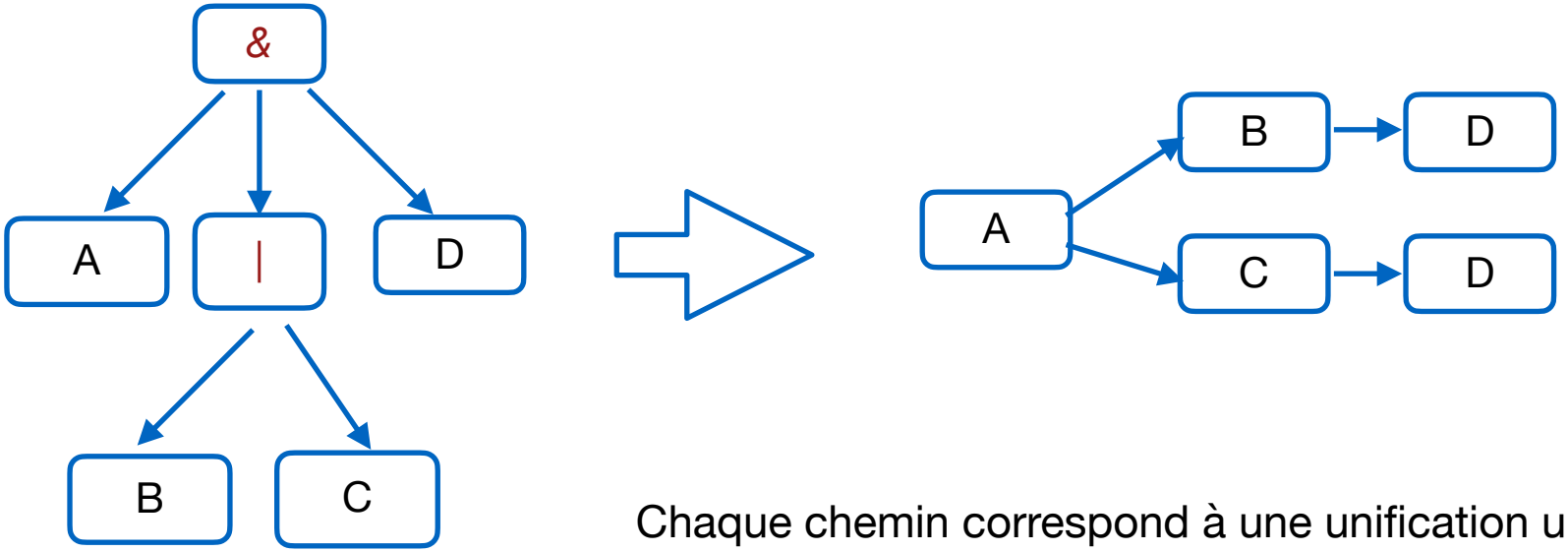
if (subj(#1,#2) & (comp(#1,#3[lemme:par]) | vcomp(#1, #3[lemme:par]))
 subj[passif=+](#1,#2).

Par exemple: *La vendeuse dit que le pain a été mangé par la souris.*

Ces règles sont exécutées comme des prédicats avec unification des variables #1,#2,#3 avec les noeuds de l'arbre. Il peut y avoir plusieurs candidats pour chacune des dépendances dans le cas de phrases complexes avec des subordonnées.

Compilation

La règle est compilée sous la forme d'un arbre: *A & (B | C) & D*
qui est ensuite transformé en une liste chainée



Chaque chemin correspond à une unification unique des variables...

Développement

Lorsque le projet commence, nous disposons déjà de modules très intéressants:

a) Des lexiques pour la plupart des langues européennes ont été constituées sous la forme de transducteurs développés par Lauri Kartunen et son équipe. On dispose ainsi d'un outil remarquable qui permet d'identifier près de 100.000 mots/s sur les PC de l'époque.
b) Un mécanisme de désambiguïsation basé sur des modèles cachés de Markov (sans API).
c) Un mécanisme de segmentations en mots et expressions basé sur les transducteurs.

Les API de ces modules étaient inexistantes ou non documentées.

Equipe

L’équipe de développement comprendra jusqu’à 20 personnes, répartis dans 3 pays: France, Japon et Portugal.

Le développement de XIP était le résultat d’une collaboration permanente avec les linguistes.

Le projet reposait sur à la fois sur des demandes spécifiques de la part des linguistes mais aussi sur des propositions de règles et de mécanismes linguistiques que je pouvais faire.

Un ingénieur gérait une architecture d’intégration qui effectuait la compilation sur toutes les plate-formes, ainsi que des tests de non régression. Ces test étaient enrichis au fur et à mesure du développement du moteur de règles.

API

XIP offrait des API dans 3 langages de programmation:

a) C++, XIP était implanté en C++ de toute façon.
b) En Java, dont une interface graphique complète.
c) En Python. Dans le cas de Python, il était d'ailleurs possible d'appeler des fonctions Python depuis les règles elles-mêmes.

Problèmes rencontrés

Nous avions plusieurs problèmes récurrents:

a) L'encodage inconstant au sein des corpus. L'UTF-8 se mélangeait allègrement avec du Latin, parfois *dans le même document.*
b) Le format d'entrée des documents, très souvent en XML à l'époque.
c) Le besoin de routines de calcul allant au-delà de règles syntaxiques.

La première solution consista à intégrer Python comme langage de script. Mais le résultat était à la fois lent et surtout impossible à débogguer.

Les règles pouvaient appeler des fonctions Python, mais Python 2.2 n'était pas stable et remonter aux sources d'un plantage s'avérait très difficile.

De plus en cas de mélange d'encodage, Python renvoyait systématiquement une erreur qui exaspérait les linguistes.

Faut pas faire ça!!!

Le mélange des encodages est une horreur à traiter.

On ne peut pas appliquer une conversion brutale à la Python...

Alors que faire?

On va supposer que tous les caractères sont *encodés en UTF-8...*

Tous les caractères <= 127, sont forcément des caractères ASCII.

Et si un caractère > 127 et n'est pas reconnu comme tel, on considère que c'est un caractère Latin et on le convertit en UTF-8.

C'est pas beau, mais ça marche pour les langues européennes de l'ouest, parce qu'en fait les séquences d'octets encodant l'UTF-8 ne correspondent à aucune séquence connue de caractères Latin pour ces langues, dans 99,99999% des cas.

La fonction...

Si l'encodage UTF-8 n'est pas reconnu, on renvoie 0:

```
char c_test_utf8(unsigned char* utf) {
    if (utf == NULL)
        return 0;

    unsigned char check = utf[0] & 0xF0;

    switch (check) {
        case 0xC0:
            return bool((utf[1] & 0x80)== 0x80)*1;
        case 0xE0:
            return bool(((utf[1] & 0x80)== 0x80 && (utf[2] & 0x80)== 0x80))*2;
        case 0xF0:
            return bool(((utf[1] & 0x80) == 0x80 && (utf[2] & 0x80)== 0x80 && (utf[3] & 0x80)== 0x80))*3;
    }
    return 0;
}
```

Voir: https://github.com/naver/tamgu/blob/master/src/conversion.cxx

L'appel de fonctions

Les règles pouvaient appeler des fonctions Python.

Mais, le résultat n'était pas à la hauteur des attentes

Dans un premier temps, j'ai commencé à définir un proto-langage de programmation dans XIP.

Puis nous avons inversé...

Nous avons défini un langage qui pouvait appeler XIP, tout en permettant l'appel de fonction de rappel depuis XIP...

Nous avons appelé ce langage KIF...

KIF

KIF pouvait non seulement charger des grammaires XIP. Il pouvait aussi les enrichir avec des règles supplémentaires.

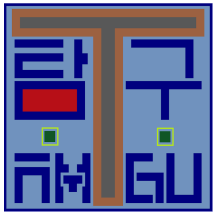
Les règles étaient aussi capables d'appeler directement des fonctions KIF...

Nous pouvions à tout moment intercepter une analyse linguistique, effectuer des tests spécifiques dans un langage proche de Python, injecter de l'information supplémentaire et laisser l'analyse continuer son cours...

Exemple

```
string grammaire='gram_no_dependency.grm';

//Chargement de la grammaire
parser p(grammaire);


//Fonction de rappel appelée depuis une règle XIP
function ranger(node n, svector v) {
    if (n.pos()=="PUNCT" and n.lemma()!='.')
        return;
    string s=n.lemma().lower();
    if ("_" in s)
        s=s.replace("_"," ");
    v.push(s);
}

//On définit la règle XIP que l'on va rajouter dynamiquement
//Elle appelle la fonction au-dessus à la fin de son exécution
string regle=@"
script:

l#1[terminal]l {
    ranger(#1, kif_exchange_data);
}
"@;

p.addendum(regle, true);
```

Tamgu 탐구



Quand XRCE est devenu Naver Labs,

***KIF** a été entièrement ré-usiné pour devenir **Tamgu** 탐구...*

Tamgu 탐구: Une combinaison de différents styles de programmation

Tamgu 탐구 combine différents types de programmation en un seul formalisme :

Impératif : Le langage ressemble à Python, mais avec une gestion des chaînes de caractères aussi puissantes que Perl et un typage fort des variables comme en Java.

Fonctionnel : Le langage offre un formalisme qui emprunte beaucoup à Haskell pour des lambdas compacts et rapides (i.e. Taskell).

Logique : Le langage offre un moteur de type Prolog

Important: Tous ces modules peuvent facilement communiquer entre eux dans un programme, puisqu'ils partagent tous les mêmes objets de base : *chaînes, vecteurs, dictionaires, nombres*....

Segmentation en mots

01100
10110
11110

La segmentation est faite à l'aide *d'expressions régulières maison,* compilées *dynamiquement* sous forme d'automates, indexés sur le premier élément de la règle...

```
//Chaines de caractères "...", tout sauf un retour chariot (%r)
\"{[\\-\"] ~%r}*\"

//Simple quote: '...', tout sauf un retour chariot (%r)
'~%r*'

//Longues chaines: @"..."@
@-\"?+\"@-

//Nombre binaire: 0b110111
0b{1 0}

//Nombre complexe: 0c12:-3i
0c({%- %+})%d+(.%d+)({eE}({%- %+})%d+):({%- %+})(%d+(.%d+)({eE}({%- %+})%d+))i

//Métarègle définissant un caractère hexadécimal
1:{%d #A-F #a-f}

//Hexadécimal: 0x4.36bca4f9165dep-3
0x%1+(.%1+)({pP}({%- %+})%d+)

//Décimal 1.234e-3
%d+(.%d+)({eE}({%- %+})%d+)
```

Ces règles peuvent être utilisées dans les programmes Tamgu탐구 et sont modifiables à loisir...

Voir: https://github.com/naver/tamgu/blob/master/src/codecompile.cxx

Arbre de la syntaxe abstraite

01100
10110
11110

Nous utilisons un deuxième jeu de règles qui lui est pré-compilé sous la forme d'un programme C++.
Ces règles combinent les tokens en un arbre.

```
########### FUNCTIONS ##############################################
instruction := %{ ;15 (blocs) %}
nonlimited := $...
arguments := declaration (%, ;6 [nonlimited^arguments])
strict := $strict
join := $joined
protecclusive := $protected^$exclusive
functionlabel := $polynomial^$function^$autorun^$thread
typefunction := (join) (protecclusive) (private) (strict) functionlabel
indexname := %[ %]
intervalname := %[ %: %]
returntype := %: %: word
space := word %@
%function := typefunction ;7 [word^plusplus^operator^comparator] %( (arguments) %) (returntype) [declarationending^instruction]
```

Voir: https://github.com/naver/tamgu/blob/master/bnf/tamgu
Voir: https://github.com/naver/tamgu/blob/master/src/codeparse.cxx

Compilateur

01100
10110
11110

Tamgu 탐구 est un *interpréteur récursif...*

Chaque noeud de l'ASA est transformé en un objet C++.

Chacun de ces objets à une méthode *Eval qu'il suffit d'exécuter.*

Tous ces objets dérivent de la même classe mère: Tamgu.

Par exemple, un objet *TamguFunction est une classe définie comme suit:*

```
class TamguFunction : public Tamgu {
public:
   vector<Tamgu*> lines;

   Tamgu* Eval() {
       Tamgu* result = aNULL;
       for (long l = 0; l < lines.size(); l++) {
           result->Release();
           result = lines[l]->Eval();
       }
     return result;
   }
}
```

Usine à objets

01100
10110
11110

Tous les objets dans Tamgu s'enregistrent dans une *usine à objets.*

Chaque objet, comme les chaines de caractères, les entiers, les flottants, les tableaux, les vecteurs, les dictionnaires etc... sont associés avec un identifiant unique.

Cet identifiant unique est associé avec un représentant de chacune de ces classes dans un immense dictionnaire.

```
global->newInstance[a_string] = new Tamgustring("", global);
global->newInstance[a_int] = new Tamguinteger(global);
global->newInstance[a_map] = new Tamgumap(global);
global->newInstance[a_vector] = new Tamguvector(global);
```

Bibliothèque

01100
10110
11110

Chaque objet expose une méthode: *Newinstance*.
Ainsi, chaque fois que l'on veut créer une variable d'un type particulier, il suffit simplement d'utiliser le *type associé à cette variable* puis appeler la méthode *Newinstance* associé à l'instance conservée dans *global->newInstance...*

```
Tamgu* nouvelle = global->newInstance[idtype]->Newinstance();
```

Créer une nouvelle bibliothèque devient enfantin.

Il suffit de créer un objet dérivant de *Tamgu,* de lui donner un identifiant unique et de lui fournir les méthodes *Eval* et *Newinstance.*

Chaque bibliothèque expose un point d'entrée unique qui permet cette initialisation:

```
extern "C" {
    Exporting bool tamgucurl_InitialisationModule(TamguGlobal* global, string version) {
       return Tamgucurl::InitialisationModule(global, version);
    }
}
```

Méthodes

01100
10110
11110

Evidemment les objets comme *string, integer* ou *vector* exposent leurs propres méthodes.

Chaque objet contient donc un dictionnaire où l'identifiant de la méthode est associé avec une méthode déclarée dans la classe C++:

```
typedef Tamgu* (Tamgucurl::*curlMethod)(short idthread, TamguCall* callfunc);

class Tamgucurl : public Tamgu {
  static map<short, curlMethod> methods;

  Tamgu* CallMethod(short idname, short idthread, TamguCall* callfunc) {
       return (this->*methods.get(idname))(contextualpattern, idthread, callfunc);
  }

   Tamgu* MethodPush(short idthread, TamguCall* callfunc) {...}
...
```

On associe l'identifiant de la méthode avec son implantation dans la classe.

```
methods[id_push] = &Tamgucurl::MethodPush;
```

Tamgu 탐구: Tendances actuelles de la programmation (Fonctionelle)

Tamgu suit la tendance actuelle d'enrichir les langages de programmation impératifs avec des capacités fonctionnelles :

- Kotlin (JVM, for android)
- Scala (JVM)
- Swift (for IOS)
- Java
- And even C++…

La plupart ont fortement emprunté des concepts à Haskell... 

et Tamgu aussi...

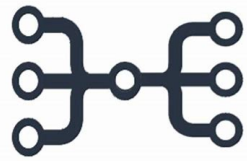
Example: code impératif...

01100
10110
11110

```
//manipulations de chaines
string u=@"Signe de la réconciliation en cours entre l'Erythrée et l'Ethiopie..."@;
println(u[12:26]); //réconciliation (automatic detection of UTF8)
println(u["en ":"entre"]); //en cours entre
u["en ":"entre"]="OH"; //"en cours entre" est remplacé par OH
ustring res=r"%C%a+" in u; //Signe
svector vs = r"%C%a+" in u; //['Signe','OH','Erythrée','Ethiopie']


//Définition de fonction
function myfunc(string s) {
    println(s);
    string e=s+".";
    return e;
}

//Définition de thread
thread mythread(int u) {
...
}

//Et aussi de classes
frame myclass {
    int i;

        function mymethod() {
                return i;
        }
}
```

Programmation fonctionnelle

Exemple: map



Haskell: *map (+1) [1..10]* *(result is [2,3,4,5,6,7,8,9,10,11])*

Tamgu *(Taskell)*:

```
vector v2 = <map (+1) [1..10]>;
```

C++:

```
vector<int> v1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
auto v2 = mapf(v1, function<int(const int&)>([](const int& i) { return i + 1; }));
```

Kotlin:

```
val v1 = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
val v2 = v1.map( { it+1 })
```

Swift:

```
let v1= 1..<11
let v2 = v1.map {1+$0}
```

Tamgu 탐구: *Traits spécifiques*

Nous avons implémenté une version assez riche de Haskell incluant :

- La compositionnalité
- L'évaluation paresseuse
- Quelques Monades (*Nothing*)
- La plupart des fonctions de base de Haskell (*map, filter, take*, etc.)
- Les structures de données Haskell
- Typages (à la fois implicite et explicite)

Le code Taskell peut fusionner en toute transparence avec le code Tamgu....

Nous fournissons également un langage de prédicat complet avec DCG qui peut également se fondre en toute transparence dans le code Tamgu.

Ces deux modules partagent les mêmes objets Tamgu de base...

Tamgu 탐구: *Traits spécifiques*

```
//évaluation paresseuse et composition
v=<map (*) . takeWhile (<10) . filter (odd) [1..] >; //[1,9,25,49,81]
```

```
//Data structures
<data Form = Circle float float float  | Rectangle float float float float>

//Data structure functions
<Surface :: Form → float>
<Surface(Circle _ _ r) = 2π×r²>
<Surface(Rectangle x y xx yy) = abs(xx-x) × abs(yy-y)>

//Calling the right surface method for each object
Surface(<Circle 10 20 30>);              // 5654.8667764616
Surface(<Rectangle 20 20 40 40>);        //400
```

Tamgu 탐구: Programmation logique

Tamgu dès aujourd'hui fournit :

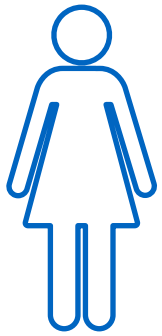
- Un moteur de prédicat entièrement intégré à Tamgu

- Règles de grammaire à clauses définies (GDC) qui peuvent être intégrées dans la structure même d'un programme.

Non seulement vous pouvez déclarer du code Prolog dans du code Tamgu directement, mais les prédicats peuvent aussi rappeler des fonctions Tamgu....

Programmation logique

Exemple: Détection de la descendance maternelle



```
//Nos relations parent, sont enregistrées dans une base de connaissance
parent("george","sam").
parent("george","andy").
parent("andy","mary").
parent("sam","christine").

femme("mary").
femme("christine").

//Then our rules
ancetre(?X,?X) :- true.
ancetre(?X,?Z) :- parent(?X,?Y),ancetre(?Y,?Z).

lineage(?X,?Q) :- ancetre(?X,?Q), femme(?Q).

//In this case, since the recipient variable is a vector, we explore all possibilities. It is directly blended into the regular code...
vector v=lineage(?X,?Z);
println(v);
```

```
[lineage("george","christine"),lineage("george","mary"),lineage("andy","mary"),
lineage("sam","christine")]
```

Programmation logique

Exemple en SWI Prolog

```
parent(george,sam).
parent(george,andy).
parent(andy,mary).
parent(sam,christine).

female(mary).
female(christine).

ancestor(X,X) :- true.
ancestor(X,Z) :- parent(X,Y),ancestor(Y,Z).

lineage(X,Q) :- ancestor(X,Q), female(Q).
```

Expressions régulières et capsules

Tamgu fournit des expressions régulières complexes pour détecter une séquence de mots dans un texte, ainsi que des lexiques de domaine.

Ces expressions régulières peuvent faire référence à :

- Un mot
- Une catégorie syntaxique
- Une étiquette (depuis un lexique ou une règle)
- Une *capsule*

Ces expressions renvoient une étiquette avec sa position dans le texte.

Capsules: *une capsule est une fonction utilisée pour comparer un élément à des sources externes de données, telles qu'une base de données ou un ensemble d'embbeddings.*

Annotation Automatique
(*Exemple basée sur ABSA: Brun et al.)*

```
a_drink ← #drink, {#drink, #PREP,#DET}*.
a_food ← #food, {#food, with, [from, (the), #Place+]}*.
```

Expressions régulières complexes

```
@drink ← "Cabernet Sauvignon".
@drink ← "mojito(s)".
@food ← sushi.
@food ← "pizza(s)".
```

Lexiques utilisateurs combinées à des lexiques de langue

string uu="It has great *sushi from Japan* and better *mojito*";

Résultat: [['a_food',[13,18],[19,23],[24,29]],['a_drink',[41,47]]]

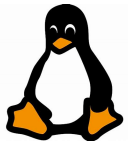
On peut dès lors utiliser ce résultat comme source d'annotation.

Annotation automatique (*Programmation de données)*
Capsule (*basée sur ABSA: Brun et al.)*

```
word2vec wrdvec(model);

// A capsule that compares a word against a focus through embeddings
function EmbeddingDistanceOK(string word, string focus) {
      if (wrdvec.distance(word, focus) >= 0.7)
            return true;
      return false;
}



//We compare our current word against the word embedding "food"
a_food ← #food, <EmbeddingDistanceOK "food">*.
```

Tamgu 탐구: Plate-formes et API

PLATE-FORMES



Tamgu 탐구 est *open-source: https://github.com/naver/tamgu*

API



Questions ?

Merci

Tamgu 탐구: Interpréteur

Tamgu 탐구 a été implémenté en C++11...

Quand vous exécutez un programme Tamgu 탐구 :

1. Tout d'abord, il est analysée sous la forme d'un arbre syntaxique (grâce à une grammaire BNF compilée en une classe C++).
2. Deuxièmement, l'arbre syntaxique est parcouru et chaque noeud est compilé en objets C++ internes qui dérivent tous de la classe racine Tamgu. Chaque objet expose un Put (pour stocker des choses) et un Get (pour obtenir des valeurs ou exécuter des choses)
3. Troisièmement, votre programme est exécuté en appelant la méthode Get de tous les objets supérieurs.

Tamgu 탐구: Cas d'usage

Tamgu est déjà utilisé dans des projets en cours:

- Annotation de texte (vue avant)
- Génération de texte
- Détection d'erreurs dans un corpus
- Détection d'adresse

Tamgu 탐구: Cas d'usage 1

Exemple génération (Programmation de données: corpus synthétique)

```
subj("manger","dame").
obj("manger","pomme").
adj("pomme","beau").
prep("de","colline").
comp("pomme","de").
avant("beau").

noun_phrase(?Noun,NP(?Det,?Adjc,?Noun)) :-
    det(?Noun,?Det),
    adj(?Noun,?Adj),
    ?Adjc is genreadj(?Noun,?Adj),
    avant(?Adj).

verb_phrase(?SNoun,?Verb,VP(?Verbc,?NP)) :- !,
    obj(?Verb,?Noun),
    conjugue(?SNoun,?Verb,?Verbc),
    noun_phrase(?Noun,?NP).

sentence(P(?NP,?VP)) :-
    subj(?Verb,?Noun),
    noun_phrase(?Noun,?NP),
    verb_phrase(?Noun,?Verb,?VP).
```

La dame mange *la belle* pomme de la colline.
La dame *mange* une belle pomme.
La dame *ne* mange *pas* la belle pomme de la colline.
Une pomme *est mangée* par la dame.

Etc…

Predicates can call functions

```
function genreadj(string n,string adj) {
   vector nounfeat=trans.lookup(n);
   for (string m in nounfeat) {
         if ("+Fem" in m)
              adj+="+Fem";
           else
              adj+="+Masc";
           break;
   }
    v=trans.lookdown(adj);
    return(v[0]);
}
```

Tamgu 탐구: Cas d'usage 2

Extension de corpus avec des phrases bruitées synthétiques (NMT)

Le but est d'étendre un corpus avec des phrases synthétiques contenant des erreurs typiques en français....

Tamgu fournit des lexiques sous la forme de transducteurs, qui peuvent être interrogés avec des *flags* de distance d'édition :

```
analyse = français.lookup(mot.lower(), 2, a_firstla_vowella_surface);
analyse = français.lookup(mot.lower(), 10, a_repetitionla_firstla_surface);
analyse = français.lookup(mot, 1, a_switchla_surface);
```

Nous avons appliqué ce programme à un corpus et extrait environ 10 000 mots différents

```
analyse = français.lookup(’resumé’, 2, a_firstla_vowella_surface);
→   ['résumé']
----------
analyse = français.lookup("trèèèèèèès", 10, a_repetitionla_firstla_surface);
→   [’très']
```

Tamgu 탐구: Cas d'usage 2

Extension de corpus avec des phrases bruitées synthétiques (NMT)

Nous voulons générer des phrases en remplaçant les mots corrects par des erreurs réelles....

Mots corrects fournis sous forme de lexique qui sera compilé dans un transducteur à la volée :

@check ← évolutions.
@check ← évolué.
@check ← évoque.

Erreurs trouvées dans le corpus

'évolutions':['evolutions'],
'évolué':['evolué'],
'évoque':['évoqque']

Tamgu 탐구: Cas d'usage 2

Extension de corpus avec des phrases bruitées synthétiques (NMT)

```
// Une règle pour détecter un mot du lexique
phrase ← #check.

annotator r; //pour accéder à la règle...

//Nous lisons un fichier ligne par ligne
for (sent in f) {
      v=r.apply(sent,true); //Nous appliquons notre règle
```

Exemple:

Ce sont des périodes … différents de culture.

[['phrase',1],['phrase',3],['phrase',13]]

['sont','périodes','différents']

Tamgu 탐구: Case d'usage 3: Adresses

Annotation d'adresses dans des textes (Denys Proux)

Comment détecter des adresses connues (à partir de 1M adresses françaises) dans un texte :

1. Tout d'abord, nous transformons ce million d'adresses en un transducteur.
2. Deuxièmement, nous mettons en place un moyen de normaliser ces adresses (par exemple, nous supprimons les mots vides).
3. Troisièmement, nous analysons le corpus avec des lexiques (méthode NTM, pour les nostalgiques).

Exemple:

```
string s= "An italian restaurant rue De L'Ancienne Prefecture  in Lieu-dit L'Usine du Pont 30 Rue de Richelieu threre 6-8 rue de Lyon la 4, Rue De L'Ancienne Prefecture.";

vector v = lexicon.parse(s);
```

['rue ancienne prefecture','lieudit usine pont','30 rue richelieu','6-8 rue lyon','4 rue ancienne prefecture']

Tamgu 탐구: Factory

The whole compiler is organised as a *factory*.

A factory is a service that enables the recording and access of various external objects…

- Each object is recorded with its *id* into the factory
- Each object exposes a method Newinstance() to create clones of itself.
- Each object exposes a method *Get()* and a method *Put()*, hence the interpreter does not need to know anything about the current object.

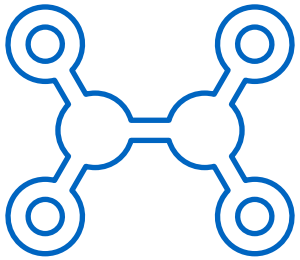
Tamgu 탐구: Extensions

Tamgu can be easily extended:

- You can extend existing basic objects in the interpreter with new methods in a couple of minutes...
- You can create your own libraries with a template that we provide

Again, there are no differences between the two approaches, they manipulate exactly the same object description...

Tamgu 탐구: Some libraries



Machine Learning:

- CRFSuite
- Wapiti (CRF)
- Word2Vec
- Liblinear

cURL *to load Web pages*

FLTK *to create GUI and graphics*

SQLite *to manage an SQL database*

libao and mp3 *to play music*

libxml2 *to handle xml files*

Tamgu 탐구: Libraries

- External libraries (.so or .dll) are based on the same object description as regular objects such a strings or containers...

- For Tamgu, an object that is loaded from an external library or pre-compiled into the main engine is *treated* exactly in the same way.

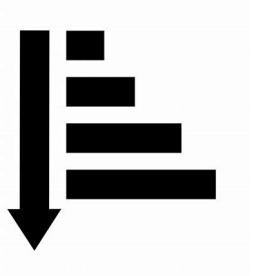
- The interpreter is totally *independent* from any specific library implementation.

Example: Handling strings

```
string u=@"Signe de la réconciliation en cours entre l'Erythrée et l'Ethiopie,
le premier vol commercial depuis vingt ans reliant les deux anciens ennemis
de la Corne de l'Afrique a décollé mercredi d'Addis Abeba."@;

println(u[12:26]); //réconciliation (automatic detection of UTF8)

println(u["en ":"entre"]); //en cours entre

ivector iv = "ré" in u; //[12,49]

u["en ":"entre"]="OH"; //Replacement with OH of "en cours entre"

ustring res=r"%C%a+" in u; //Signe

//['Signe','OH','Erythrée','Ethiopie','Corne','Afrique','Addis','Abeba']
svector vs = r"%C%a+" in u;
```

Functional Programming

Examples



```
<fastsort([]) = []>  //if the list is empty, we return an empty "list"
<fastsort([fv:v]) =  mn &&& fv &&& mx where //we merge the different lists...
    let mn = <fastsort . filter (<=fv) v>,   //smaller than fv
    let mx = <fastsort . filter (>fv) v>     //larger than fv
>

vector vv=[1,5,7,2,8,0];
vector v=fastsort(vv);
```

v is [0,1,2,5,7,8]

Logical programming

Example DCG

```
sentence(s(?NP,?VP)) --> noun_phrase(?NP), verb_phrase(?VP).
noun_phrase(np(?D,?N)) --> det(?D), noun(?N).
verb_phrase(vp(?V,?NP)) --> verb(?V), noun_phrase(?NP).
det(d("the")) --> ["the"].
det(d("a")) --> ["a"].
noun(n("bat")) --> ["bat"].
noun(n("cat")) --> ["cat"].
verb(v("eats")) --> ["eats"].

//we generate all possible interpretations...
vector vr=sentence(?Y,[],?X);
```

Results:

```
sentence(["the","bat","eats","the","bat"],[],s(np(d(the),n(bat)),vp(v(eats),np(d(the),n(bat)))))
sentence(["the","bat","eats","the","cat"],[],s(np(d(the),n(bat)),vp(v(eats),np(d(the),n(cat)))))
sentence(["the","bat","eats","a","bat"],[],s(np(d(the),n(bat)),vp(v(eats),np(d(a),n(bat)))))
etc.
```

Tamgu 탐구: Implementation

How do you implement a programming language in the first place?

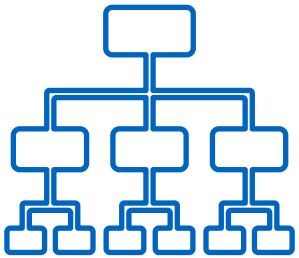
You start with a BNF (Backus–Naur Form).

This is a grammar that describes your language.

It is compiled into a C++ class...

This C++ class is used to:

- parse your program
- build a syntactic tree
- Which is then compiled into C++ *objects*



Tamgu 탐구: BNF

```
############ FUNCTIONS ##################################

nonlimited := $...

arguments := declaration (%, ;6 [nonlimited^arguments])

strict := $strict
join := $joined

protecclusive := $protected^$exclusive

functionlabel := $polynomial^$function^$autorun^$thread

typefunction := (join) (protecclusive) (private) (strict) functionlabel
```