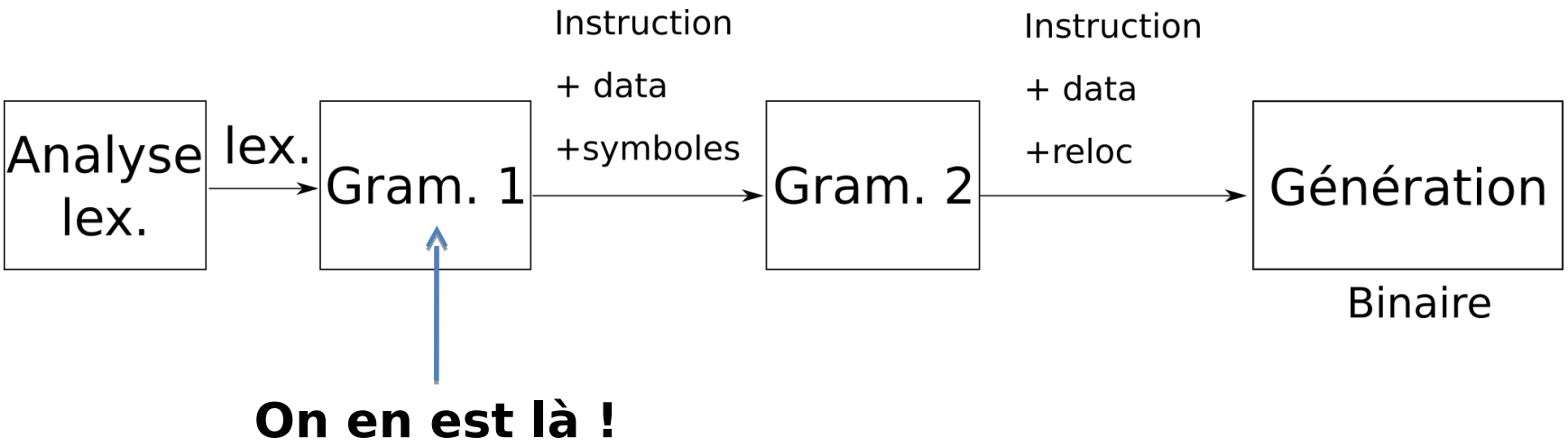
Assembleur MIPS
Incrément 2

Notes de BE

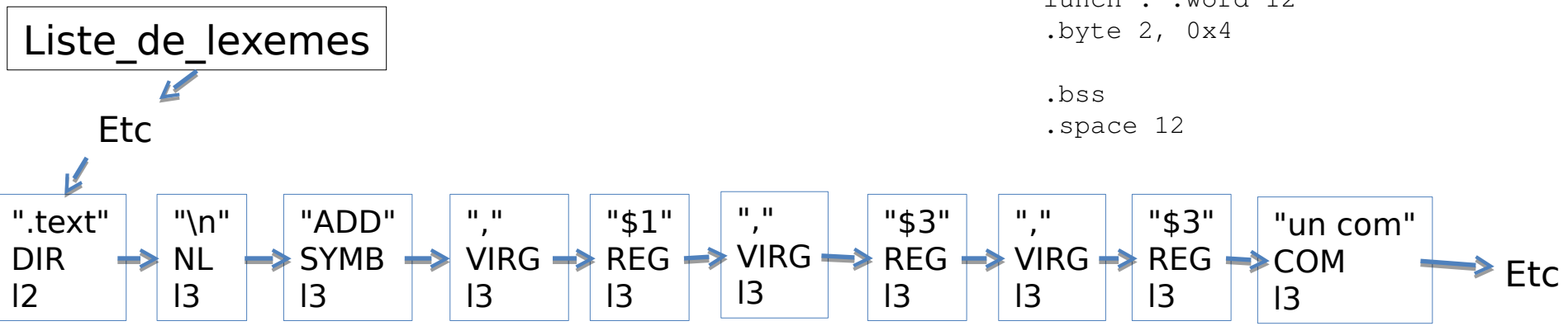
Date de rendu incrément 2 SICOM et SEI :
cf. le site web


Analyse lex.
lex.
Gram. 1
Instruction
+ data
+symboles
Gram. 2
Instruction
+ data
+reloc
Génération
Binaire
On en est là !

```
.set noreorder
.text
ADD $1, $2, $3   #un com
LW $6, -200($7)

.data
lunch : .word 12
.byte 2, 0x4

.bss
.space 12
```

Rappel étape 1 : analyse lexicale

```
.set noreorder
.text
ADD $1, $2, $3    #un com
LW $6,-200($7)

.data
lunch : .word 12
.byte 2, 0x4

.bss
.space 12
```

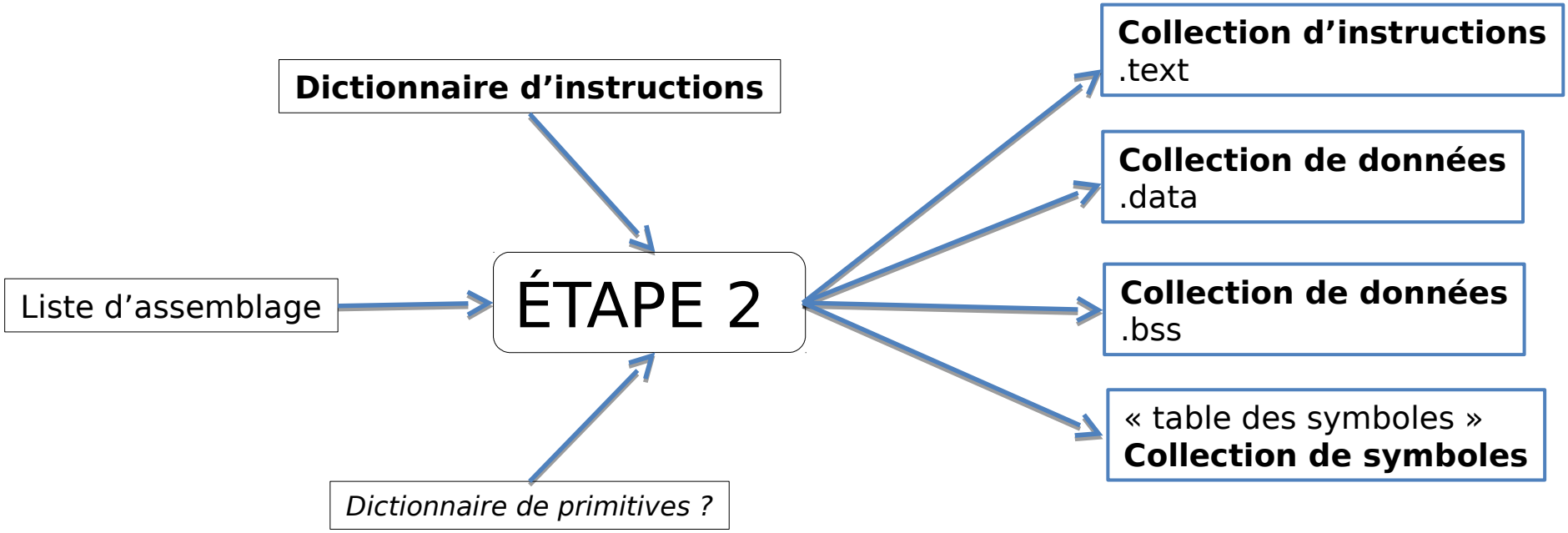


[DIRECTIVE	]	.set	ligne 1
[SYMBOLE	]	noreorder	ligne 1
[DIRECTIVE	]	.text	ligne 2
[NL	]		ligne 2
[SYMBOLE	]	ADD	ligne 3
[REGISTRE	]	$1	ligne 3
[VIRGULE	]	,	ligne 3
[REGISTRE	]	$2	ligne 3
[VIRGULE	]	,	ligne 3
[REGISTRE	]	$3	ligne 3
[COMMENTAIRE	]	un com	ligne 3

..... etc

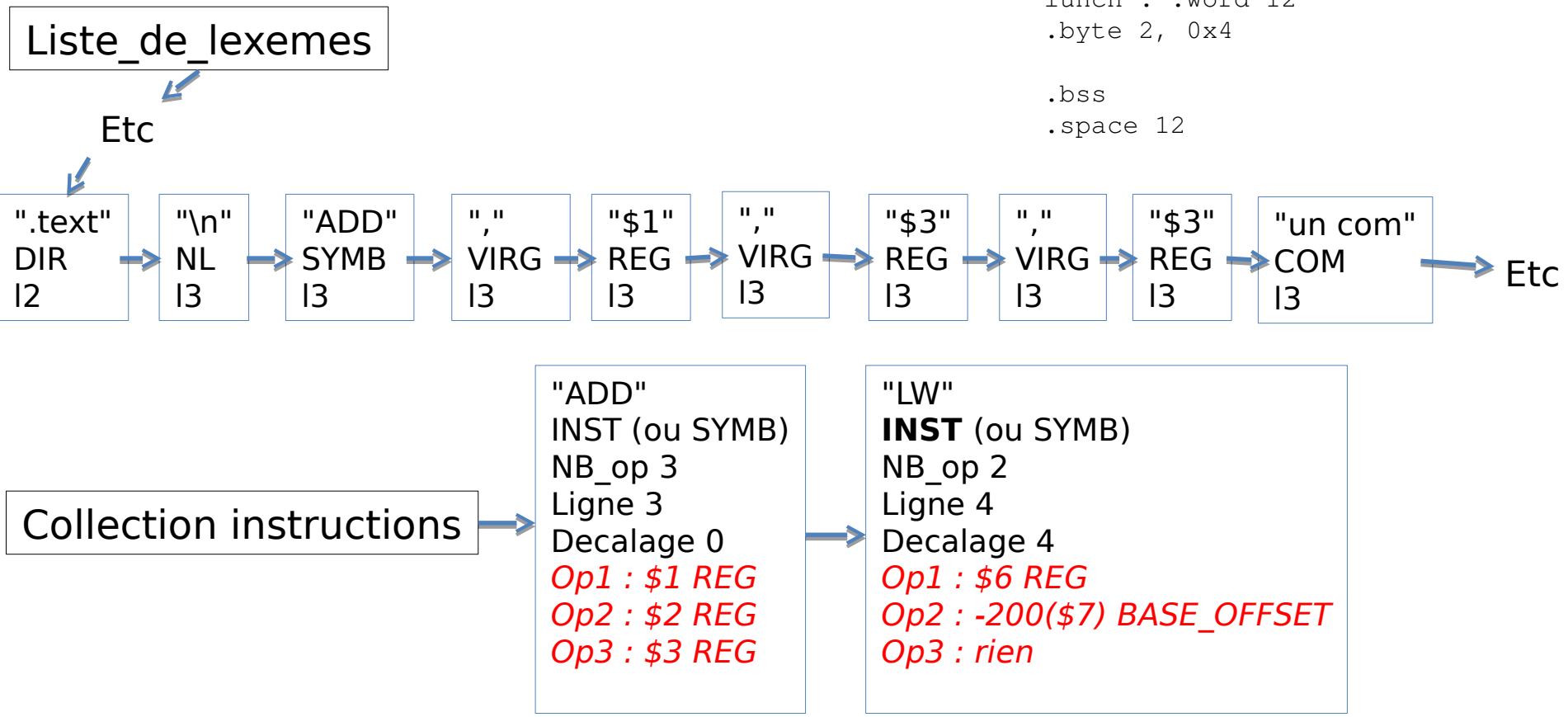
Étape 2 : analyse grammaticale (ou « syntaxique ») 1 :

Passer d'une collection de lexèmes à 4 collections: collection **d'instructions (.text) , de données .data , de données .bss et de symboles**.
Section .text : en vérifiant si les instructions existent et ont le bon nombre d'opérandes, au moyen d'un *dictionnaire d'instruction.* Mais PAS de vérification de types d'opérandes et de leur valeur : c'est pour l'étape suivante.
Section .bss et .data : en vérifiant que les primitives sont correctement utilisées, que leurs opérandes sont correctes (bon types, valeurs adéquates…) et en extrayant les valeurs de ces opérandes.
Collection de symboles : stocker toutes les déclarations d'étiquettes et leur position (adresse) dans la section où elles sont déclarées.



Collection d'instructions ?

```
.set noreorder
.text
ADD $1, $2, $3    #un com
LW $6,-200($7)

.data
lunch : .word 12
.byte 2, 0x4

.bss
.space 12
```



```
.set noreorder
.text
ADD $1, $2, $3    #un com
LW $6,-200($7)

.data
lunch : .word 12
.byte 2, 0x4

.bss
.space 12
```

Liste_de_lexemes

Etc

".text" DIR l2 → "\n" NL l3 → "ADD" SYMB l3 → "," VIRG l3 → "$1" REG l3 → "," VIRG l3 → "$3" REG l3 → "," VIRG l3 → "$3" REG l3 → "un com" COM l3 → Etc

Collection instructions →

"ADD"
INST (ou SYMB)
NB_op 3
Ligne 3
Decalage 0
Op1 : $1 REG
Op2 : $2 REG
Op3 : $3 REG

→

"LW"
INST (ou SYMB)
NB_op 2
Ligne 4
Decalage 4
Op1 : $6 REG
Op2 : -200($7) BASE_OFFSET
Op3 : rien

Comment gérer ce type d'opérande **Base_Offset** ? Une liste ? Autre?

Pour cet incrément : Stocker les opérandes est optionnel

Reconnaître les types des opérandes est optionnel

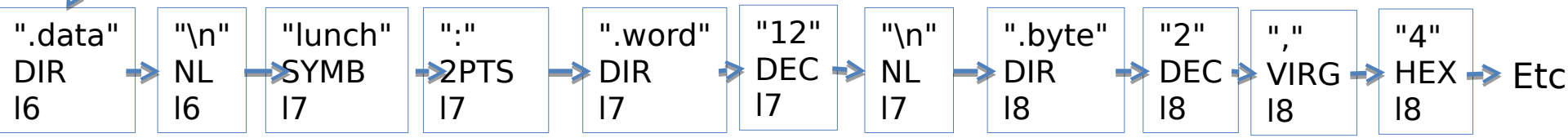
Types opérandes (REG, BASE_OFFSET, NOMBRE, SYMBOLE, autre

Collection de données ?

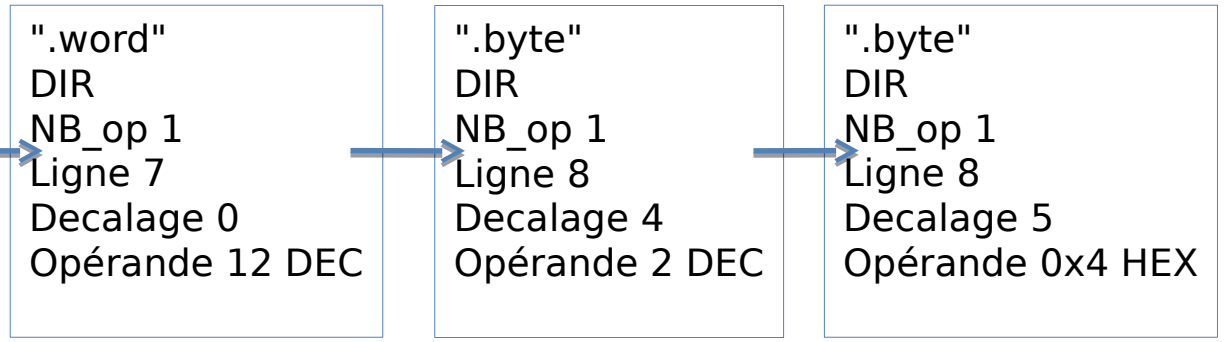
```
.set noreorder
.text
ADD $1, $2, $3    #un com
LW $6,-200($7)

.data
lunch : .word 12
.byte 2, 0x4
.word lunch

.bss
.space 12
```

Liste_de_lexemes

Etc

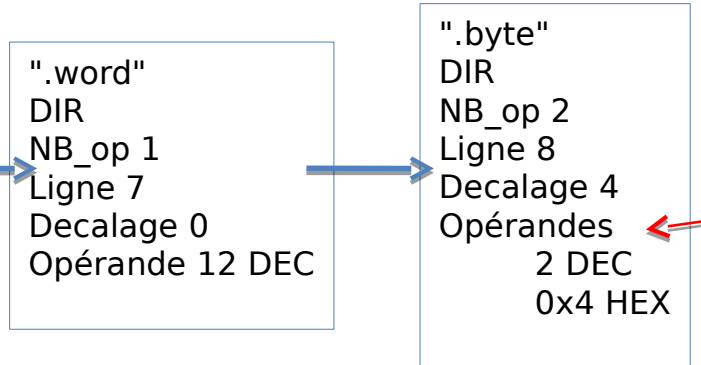


Collection données 1
Section DATA



Ou bien….

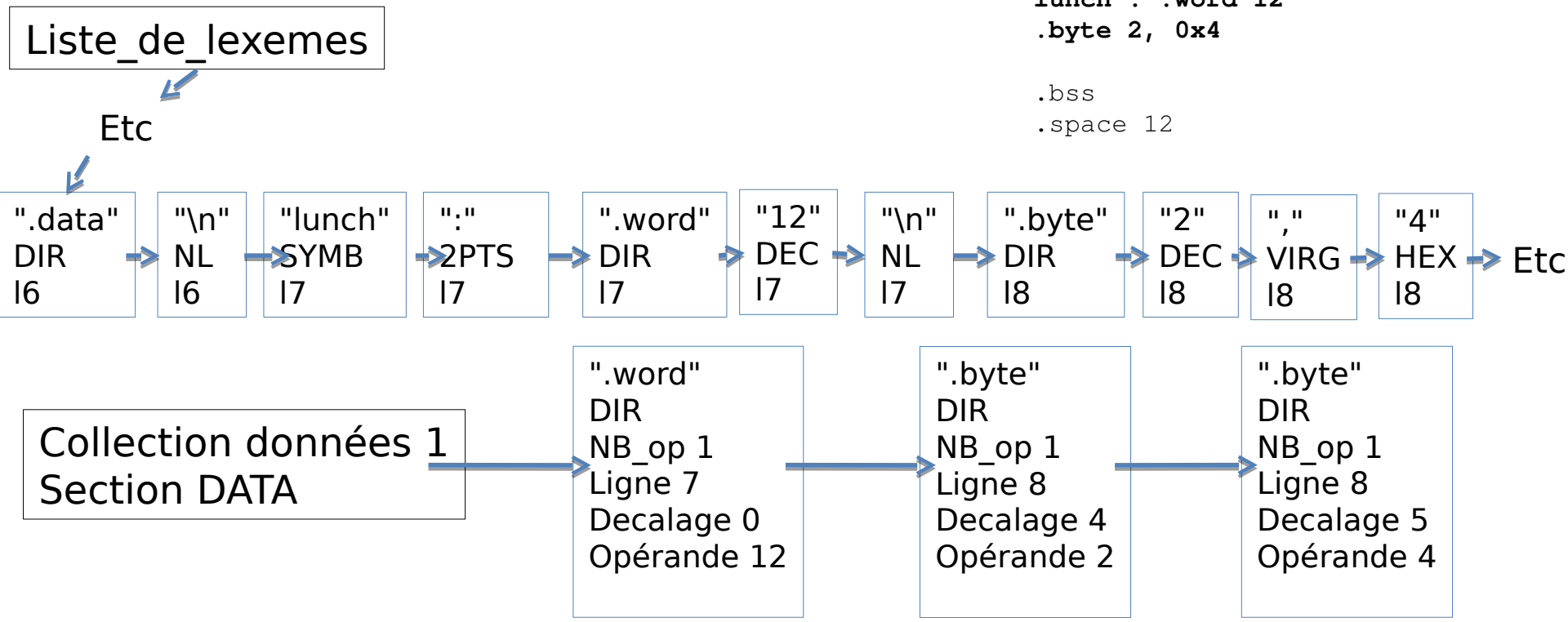
Collection données 1
Section DATA



Comment gérer un nb quelconque d'opérandes ?

Collection de données ?

```
.set noreorder
.text
ADD $1, $2, $3    #un com
LW $6,-200($7)

.data
lunch : .word 12
.byte 2, 0x4

.bss
.space 12
```



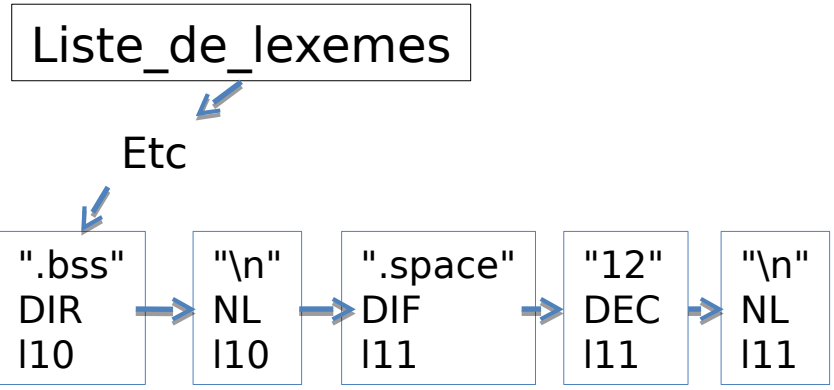
Pour cet incrément :

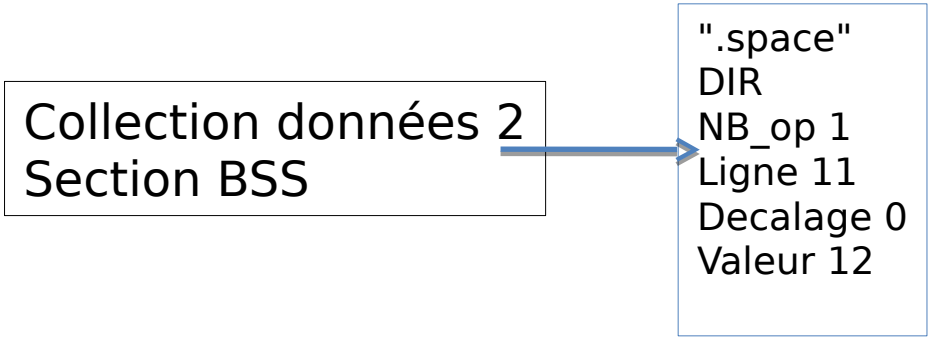
Analyser (type, valeur) les opérandes des primitives des sections DATA et BSS n'est PAS optionnel

(car plus facile que pour les instructions)

Types possible des opérandes : CHAR, INT, UNSIGNED INT, SYMBOLE, CHAINE DE CHAR *(pour .asciiz)*. Autre ?

Collection de données ?

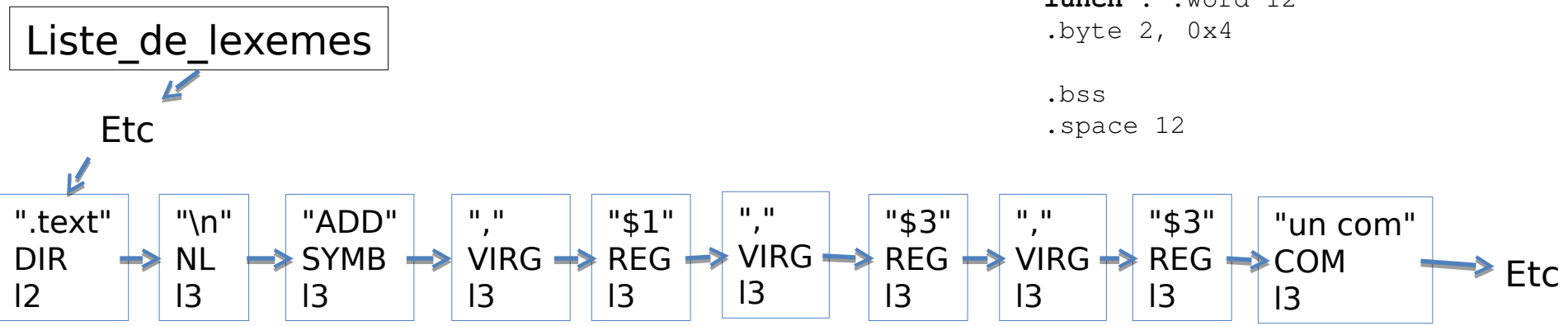


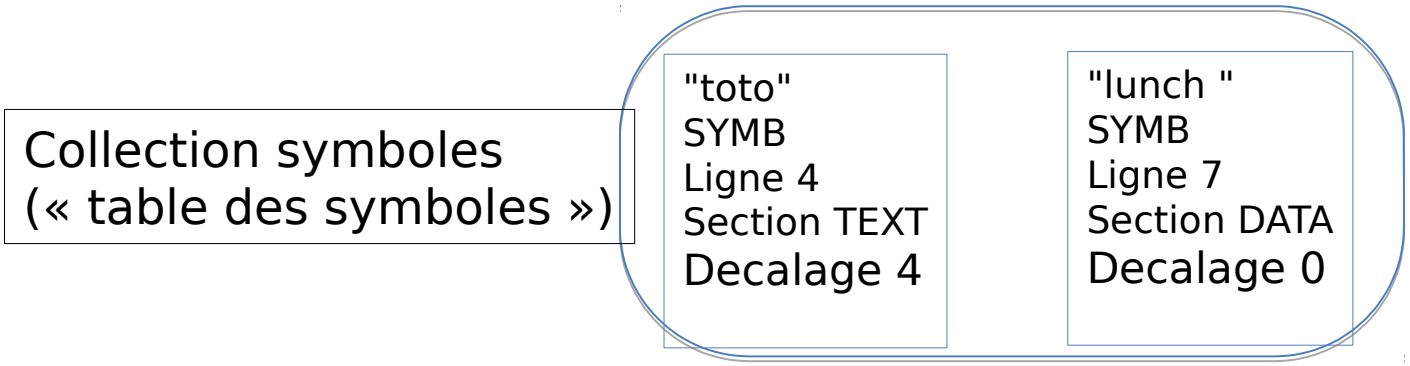



```
.set noreorder
.text
ADD $1, $2, $3    #un com
LW $6,-200($7)

.data
lunch : .word 12
.byte 2, 0x4

.bss
.space 12
```

Table des symboles ?

```
.set noreorder
.text
ADD $1, $2, $3    #un com
toto: LW $6,-200($7)

.data
lunch : .word 12
.byte 2, 0x4

.bss
.space 12
```





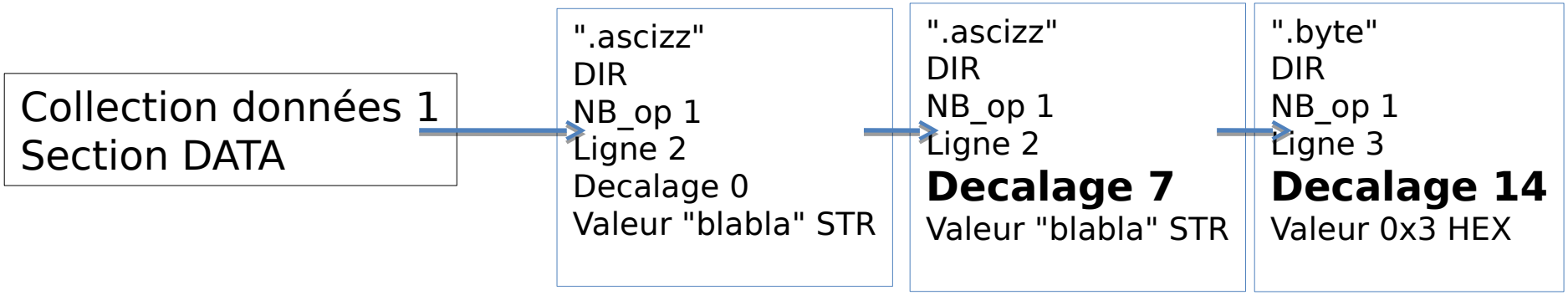
Pour cet incrément, on ne traite que les définitions d'étiquettes (suivie de ":")

Pas les usages, comme :

```
BEQ $t0 , $t1 , toto
```

Cas de .asciiz

```
.data
.asciiz "blabla", "blibli"
.byte 0x3
```



Attention au caractère de terminaison `'\0'`

Primitives valides dans les sections .data et .bss ?

```
.byte
.word
.asciiz
.space
```

Acceptent plusieurs paramètres

Exemple :

```
    .byte 4, 9, -5, 0x1A
    .asciiz "ah","que \"coucou\n"
    .word uneEtiquette, -12652
```

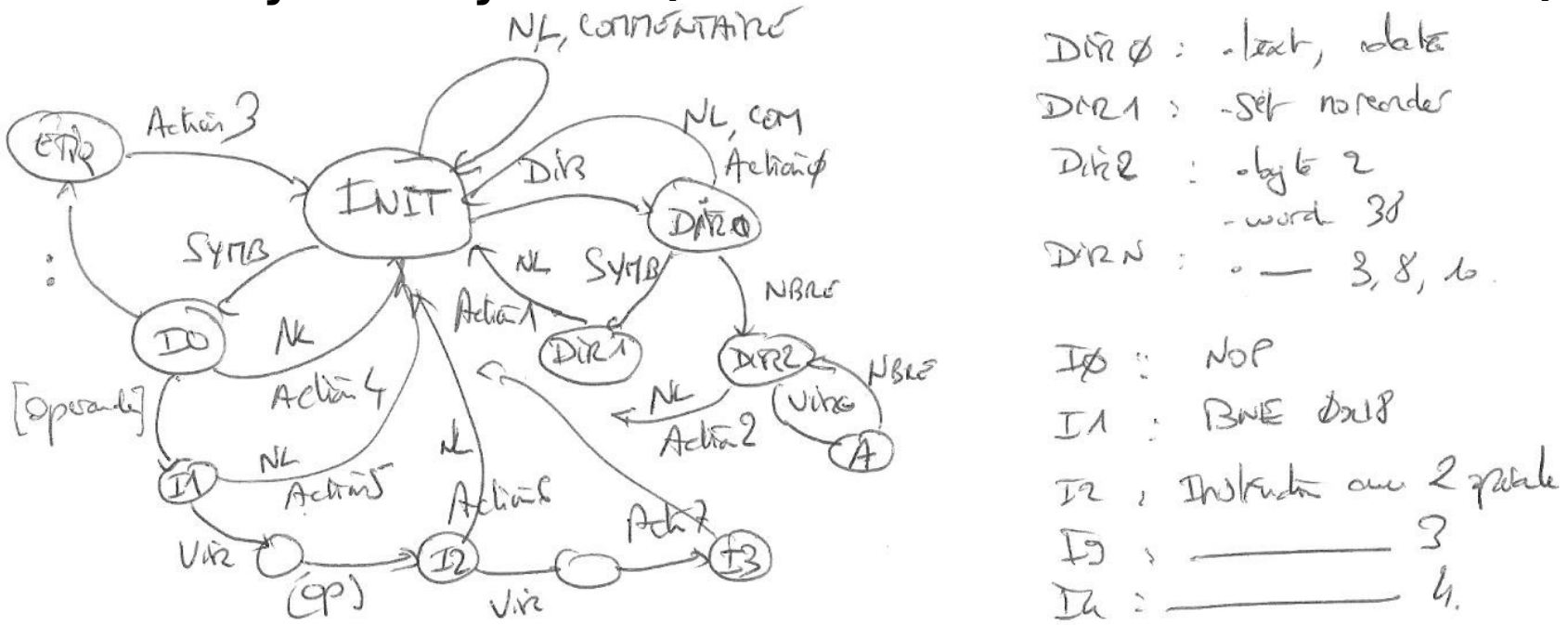
```
.byte
.word
.asciiz
.space
```

Sont des primitives acceptées dans la section .data

```
.space
```

Est seule acceptée dans la section .bss

Comment analyser la syntaxe (afin de construire les collections ?)



Action0 : c'est un .text ou un .data, ça ne sert qu'a indiquer qu'on ajoutera dans la collection des data ou text. Donc, une variable mise à jour qui sert ensuite aux autres actions

Action1 : un .set noreorder : on ne fait rien,

Action2 : un .byte 2 par exemple : on stocke les 2 composantes dans un chaînon de la collection des data. On vérifie que le symbole est bien une directive et que le nombre de paramètres est bon.

Action2_bis : un .asciiz ''bonjour'' même mécanisme mais avec une chaîne de caractères.

Action3 : une étiquette : on la stocke dans la table des étiquettes en vérifiant qu'elle n'existe pas déjà (double définition)

Action5 : une instruction avec un opérande : on stocke les 2 composantes dans un chaînon de la collection des instructions. On vérifie grâce au dictionnaire que le symbole est bien une instruction gérée et que le nombre de paramètres est bon (suffit de compter les « , »).

Comment analyser la syntaxe (afin de construire les collections ?)

=> Un automate d'état fini

Durant l'exécution de cet automate, il faudra notamment :
Une variable qui indique si on est en train de travailler sur la section TEXT DATA ou BSS (« macro-état »)
Pour chaque section, une variable qui indique quelle est l'adresse en cours de traitement dans la section
(par exemple, pour .text : +=4 à chaque nouvelle instruction)

Pour la section .text : vérifier que les symboles des instructions sont valides ainsi que le nombre d'opérandes de chaque instruction

Pour la section .data et .bss : vérifier que les symboles des directives sont valides
ainsi que le nombre d'opérande, leur type et leur valeur…
et extraire les valeurs des opérandes

Gérer les erreurs de syntaxe et de grammaire… !

Structure pour stocker une instruction (section .text) ?

- L'instruction ou son lexème
  (ou mieux un pointeur vers la définition de l'instruction dans le *dictionnaire*)
- Le numéro de ligne dans le code
- L'adresse relative dans la section (décalage depuis début section)
- Le nombre d'opérande trouvée dans le code (doit être conforme à la spécification de l'instruction)
- OPTIONNEL : les opérandes
  - *Rappel* : on analysera pas les opérandes durant cet incrément, car facile pour -2 ou 0xFA2, mais plus délicat pour par exemple 1000($7).
  - Mais il serait bien d'extraire les opérandes de la liste de lexèmes, pour les stocker dans les instruction de la collection d'instructions.

    Quelle structure de donnée ?
    - Un tableau des opérandes ? Mmmhhh
    - Une liste chainée des opérandes ? Mmmhh

    ⇒ peut être un *tableau de liste de lexèmes, de taille 3 car il y a au maximum 3 opérandes par instruction*

Structure pour stocker une donnée (sections .data ou .bss) ?

• La directive ou son lexème
• Le numéro de ligne
• L'adresse relative dans la section (décalage depuis début section)
• (le nombre d'opérandes… ou pas… suivant ce qu'on décide pour les primitives acceptant plusieurs opérandes)
• le **type** et la **valeur** de l'opérande (ou de chaque opérandes)

Comment stocker les valeurs des opérandes, sachant qu'elles peuvent être de divers types ?

- Quels sont les types possibles ?
  - char <= pour .byte
  - int <= pour .word
  - char * <= pour asciiz
  - char * <= pour le cas ou l'opérande est un symbole (étiquette)
  - unsigned int <= pour .space (suivi d'une taille en octet)
  - et peut être aussi : unsigned char (octet non signé) , etc… ???

Utiliser une UNION, cf cours 1A 2ème semestre PET et explications en séance

Structure pour stocker les données (sections .data ou .bss)

Liste chaînée de déclarations de données ?

Structure pour stocker un symbole (table des symboles)

• une chaîne : le symbole
• Le numéro de ligne *de définition dans le code, et non pas d'utilisation*
• La section où ce symbole est défini
• L'adresse relative dans la section (décalage depuis début section)

Comment stocker les symboles
(quelle structure de donnée pour la table des symboles) ?

Utiliser un tableau de symboles ?
Utiliser une liste de symboles ?
Comment faire mieux ?

L'indexation est-elle importante ?
Quelles sont les opérations importantes sur la collection de symboles ?
Le même symbole peut-il être défini plusieurs fois ?
Et donc, quel type abstrait serait adéquat ??

Stocker les informations sur les instructions : un dictionnaire d'instruction

Type structuré « définition d'instruction »

Symbole de l'instruction (attention : BEQ et beq c'est la même chose

=> `strcasecmp(…)`)

Type de l'instruction (R, I, J ou P ; P pour « pseudo instruction »)

Nombre d'opérandes de l'instruction

Par exemple :

```
typedef struct { char * symbole;  char type ; int nb_op ;} inst_def_t;
// Et pour créer une variable représentant en mémoire le dictionnaire : tableau de définition d'instructions :
inst_def_t   dicoDefInstruction[ ] ;
int nbDefInstructions;
```

Comme on aura beaucoup de types d'instructions à gérer (de plus en plus…), on va stocker les définition d'instructions dans un fichier texte : le « dictionnaire d'instructions »

Par exemple :

Fichier dicoInstruction.txt

```
4
ADD R 3
ADDI I 3
LW I 2
NOP P 0
```

Comment charger ce fichier en mémoire ?

Un dictionnaire pour reconnaître les instructions ?

Comment charger ce fichier en mémoire ? Quelque chose comme :

```
typedef struct { char * symbole;  char type ; int nb_op ;} inst_def_t;
// Charge le fichier dictionnaire nomFichierDico
// Retourne un pointeur sur le tableau dictionnaire
// stocke le nb d'instructions dans *p_nb_inst
inst_def_t * lect_dico_int(char* nomFichierDico, int* p_nb_inst) {
    FILE *f1;
    int i;
    char s1[512];
    inst_def_t  * tab;
    f1=fopen(nomFichierDico,"r");
    if (f1==NULL)
           return NULL;
    if( fscanf(f1,"%d",  p_nb_inst) != 1){
        return NULL;
    }
    tab=calloc(*p_nb_int,sizeof(*tab));
    if(tab == NULL) ... Etc ...
    for (i=0; i<*p_nb_int; i++) {
        if(fscanf(f1,"%s %c %d",s1,&(tab[i].type),&(tab[i].nb_op)) != 3) {
            free(tab);
            return NULL;
        }
        tab[i].nom=strdup(s1);
    }
    fclose(f1);
    return tab;
}
```

Fichier texte dictionnaire

```
4
ADD R 3
ADDI I 3
LW I 2
NOP P 0
```

Et pourquoi pas un second dictionnaire pour reconnaître les primitives ?

Pas si facile… car :

Les types acceptés pour les opérandes sont variés

eg : .word accepte un entier signé sur 4 octets, ou un symbole (étiquette)

Le nb d'opérandes accepté est varié (0 .pour `.text`, 1 pour `.set`, infini, …)

L'endroit où peut apparaître la primitive devrait être spécifié :

.set noreorder ne devrait apparaître qu'au début (avant tout le reste)

.space peut apparaître dans la section DATA ou BSS

.data peut apparaître n'importe où…

etc.

Des cas particuliers

.set noreorder

Pas si utile…

Car le nombre de primitive est très réduit, tout de même…

⇒ Soit on monte un 2^{e} dico pour les primitives… mais bof…

⇒ Soit on gère au cas par cas (une fonction C par primitive ???)

Des listes génériques ?

Vous avez déjà des listes de lexèmes :

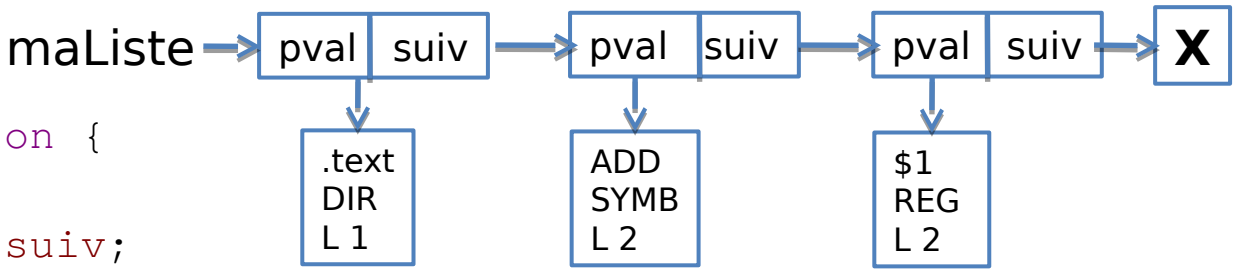
On va avoir besoin de listes d'instructions, des listes de données, (de listes de symboles), … Et ça ne va pas s'arrêter là…

Que faire ?

Un type de liste (un module ListeDeMachin.h ListeDeMachin.c) par type de chose qu'on veut mettre dans les listes ?

Autre solution, pas « nécessaire » mais rusée : listes génériques

Ou : *Liste dont les éléments sont des « pointeurs vers n'importe quoi »*



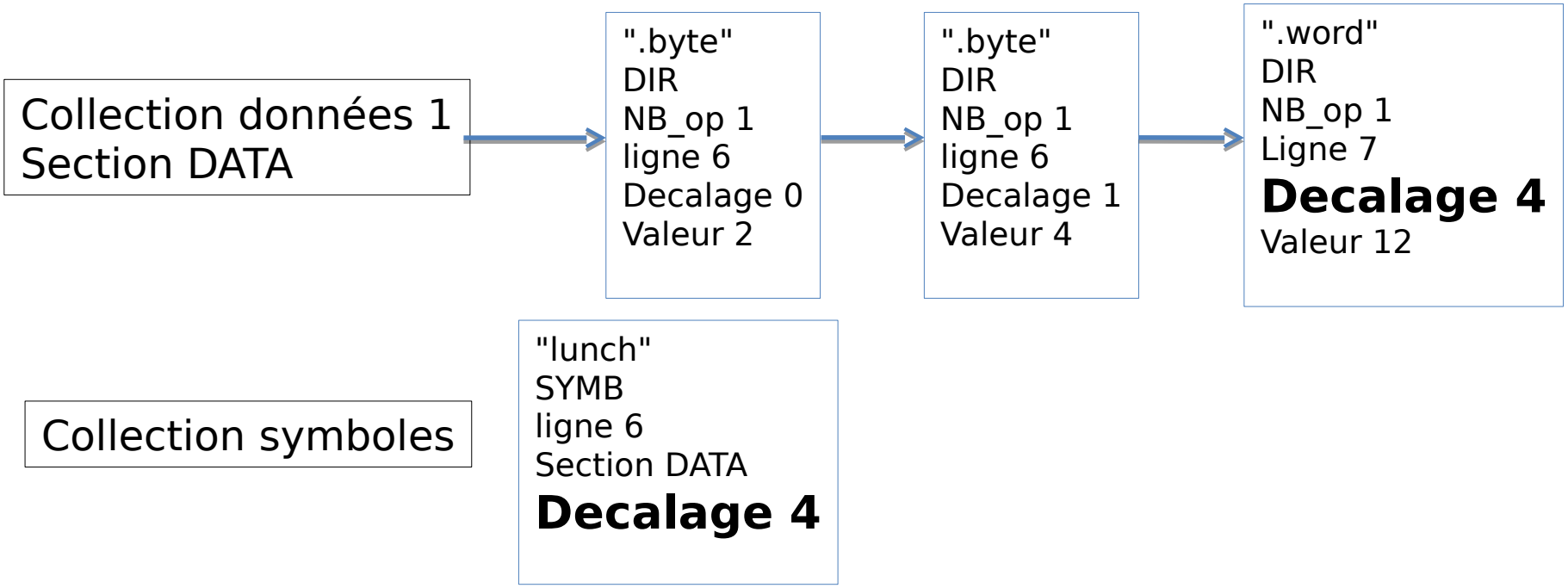

```
typedef struct maillon {
     void*  pval;
     struct maillon *suiv;
} *LISTE_GENERIQUE;

LISTE_GENERIQUE maListe ;
```

Explications poursuivies en BE … ou pas

Quelques compléments…

Attention : `.word` est toujours alignée sur 4 octets !

```
.data
.byte 2, 0x4
lunch : .word 12
```



Quelques compléments… vers des choses un peu « tricky »

Adresses des symboles dans la table des symboles : pas toujours si simple…

```
.data
.byte 2, 4
toto :
titi :
.word -12
```

Attention :
dans ce cas, les adresses de `toto` et `titi ne` sont connues UNIQUEMENT après avoir détecté .word et procédé à l'alignement !

Piste de solution :

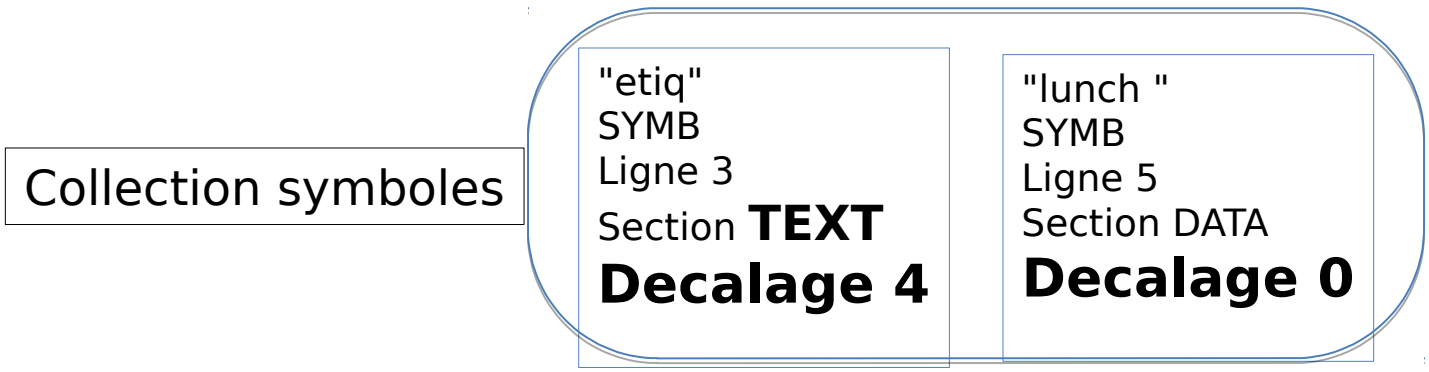
1/ lorsqu'on traite les déclarations d'étiquette `toto:` et `titi:` mettre temporairement `tata et titi` dans une « liste d'étiquette à créer dans la table des symboles »

2/ lorsqu'on traite `.word 12`, vider la liste et ajouter `tata et titi` avec leur adresse (maintenant connue….) dans la table des symboles

Quelques compléments… vers des choses un peu « tricky »

Attention : `.text .data .bss` peuvent apparaître plusieurs fois dans le code

```
.text
ADD $1, $2, $3   #un com
etiq:

.data
lunch: .word 12

.text
LW $6,-0x200($7)

.bss
.space 12

.text
ADDI (etc…)
```



Quelques compléments… vers des choses un peu « tricky »

Attention : il n'est pas interdit qu'une étiquette ait le nom d'une instruction
(par exemple)

```
.text
add :          #une étiquette nommée « add » ? Eh bien, c'est possible…
add $1, $2, $3
J add

# oui, bon, ce code est bien absurde… mais il est valide !
```

« idéalement », il faudrait donc que l'automate d'état fini mis en œuvre pour analyser la liste de lexème et construire les 4 listes attendues pour cet incrément soit « résistant » à ce type de cas…

Quelques compléments…

Quelques exemples de cas d'erreur à traiter (… ou pas…) durant cet incrément

(merci à votre bel automate…)

```
.text
ADD $2, $3
NOP toto
KESAKO $2, $3, $4
ADD $2, $3, 12            #=> erreur, mais non détectée à cet incrément
ADDI $2, $3, 0xFFFFFF     #=> erreur, mais non détectée à cet incrément

.data
.asciiz 12
.byte 0xFFFF
.byte -129
.byte 129                 # c'est une erreur, ça, ou pas ?
.word 0xFFFFFFFF          # c'est une erreur, ça, ou pas ?
.word -135378953789453789543785378
.byte 12, -5,

.bss
.byte 12
.space -23
```

Rappel : pour tester votre programme,
écrire des fichier .s pour les cas « qui marchent »
… mais aussi pour les cas d'erreur !