

Ch. 7

Assemblage et Compilation

B. Quoitin
(bruno.quoitin@umons.ac.be)

Tables des Matières

- Compilation
 - Analyse lexicale et syntaxique
 - Génération de code
- Assemblage
- Edition de liens

Assemblage et Compilation

- **Objectifs**

- Travailler en langage machine est fastidieux et sujet à des erreurs. Le langage machine n'est clairement pas destiné à la création de grands programmes composés de plusieurs millions d'instructions. Aujourd'hui pour la plupart des programmes, des **langages de haut niveau** tels que C, C++, java, python et bien d'autres sont utilisés.
- L'objectif de cette section est de faire le pont entre les langages de programmation utilisés habituellement et l'architecture du processeur ainsi que son jeu d'instructions.
- Pour arriver à cet objectif, cette section introduit les principes qui vont permettre qu'un programme écrit en langage de haut-niveau soit
 - **traduit** en une suite d'instructions en langage machine,
 - **chargé** en mémoire
 - et finalement **exécuté** par un processeur.

Assemblage et Compilation

- **Compilation**

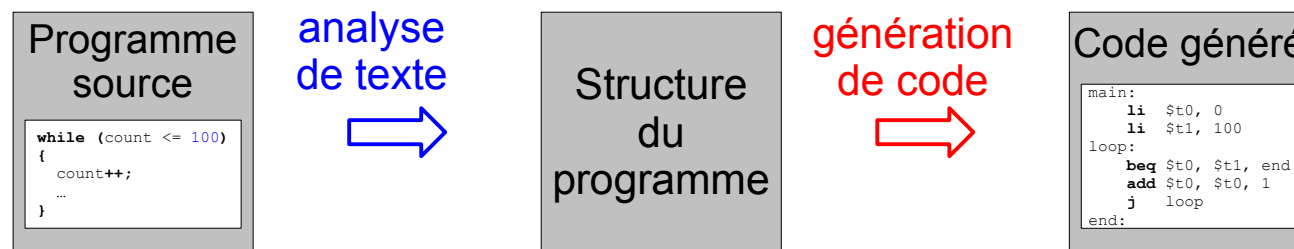
- La **compilation** est le processus de *traduction* d'un langage source vers un langage cible.
 - langage source = langage de haut-niveau (p.ex. C)
 - langage cible = langage de plus bas niveau (p.ex. langage d'assemblage).
- Le **compilateur** (*compiler*) est un programme qui effectue la compilation.
 - Un compilateur est généralement spécifique à un langage source.
 - Un compilateur peut être prévu pour une seule architecture destination ou il peut être utilisé pour générer du code pour des architectures différentes. On parle alors de **compilateur re-ciblable** (*re-targetable compiler*).

Note: la compilation est un sujet complexe qui fera l'objet d'un cours complet donné en BAC3 par Véronique Bruyère.

Assemblage et Compilation

- **Compilation**

- Afin de convertir un programme en langage de haut-niveau vers un langage d'assemblage, un compilateur doit effectuer plusieurs opérations.



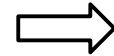
- **Analyse du texte du programme source** pour en trouver la structure: les expressions arithmétiques et logiques, les structures de contrôles (if, if/else, boucles), etc.
- **Génération d'une suite d'instructions** : allocation des registres et de la mémoire pour les résultats intermédiaires (problème d'optimisation) + ordonnancement des instructions (problème d'optimisation)

Assemblage et Compilation

- **Analyse lexicale**

- Le programme est d'abord découpé en petits morceaux élémentaires appelés **lexèmes** (*tokens*): mots-clés et opérateurs du langage, identifiants, littéraux, etc. Cette opération est appelée **analyse lexicale** (*lexical analysis*) et est réalisée par un analyseur lexical (*tokenizer*).

```
while (count <= 100) {  
    count++;  
    ...  
}
```



Suite de
lexèmes

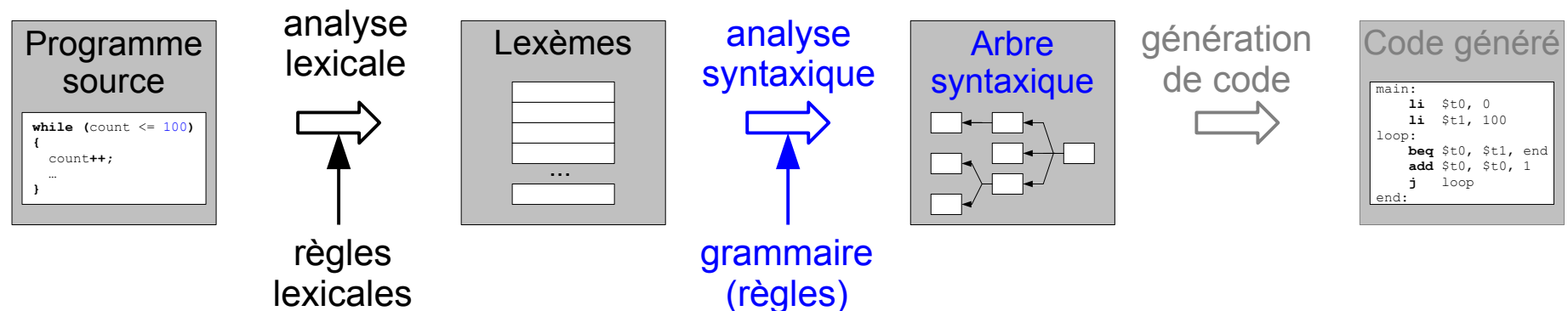
while
(
count
<=
100
)
{
count
++
;
...
}

Note : exemple tiré de **The Elements of Computing Systems: Building a Modern Computer From First Principles**,
N. Nissan and S. Schocken, MIT Press, 2005.

Assemblage et Compilation

- **Analyse syntaxique**

- La **syntaxe du langage** décrit la structure des déclarations de variables, des expressions, des instructions de contrôle (if, if/else, ...), des déclarations de fonctions, etc.
- Cette structure est formalisée dans la **grammaire du langage**, une suite de règle de transformation qui permettent de grouper les lexèmes produits par l'analyse lexicale.
- Cette opération est appelée **analyse syntaxique** (*syntactic analysis*) et est réalisée par un analyseur syntaxique (*parser*). Le résultat est généralement représenté sous la forme d'un arbre.



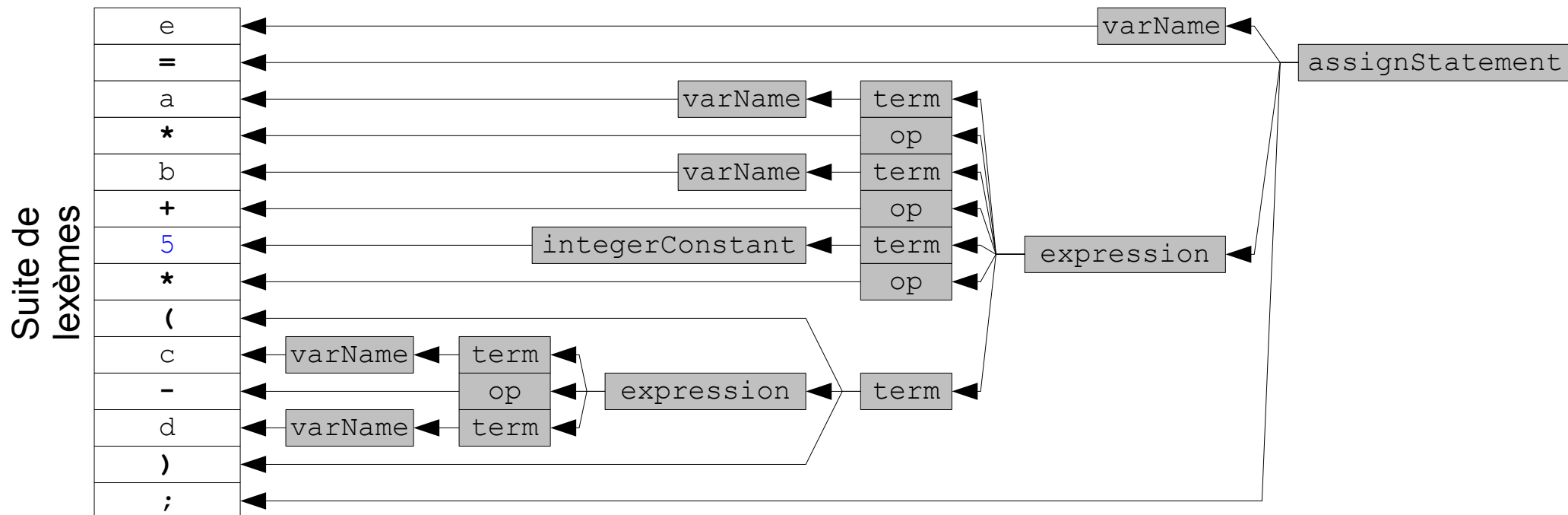
Assemblage et Compilation

- **Example**

```
e= a*b+5*(c-d) ;
```

Grammaire du langage

```
statement: ...
whileStatement: ...
ifStatement: ...
sequence: ...
assignStatement: varName '=' expression ';'
expression: term (op term)*
term: integerConstant | stringConstant | varName
      | '(' expression ')' | ...
op: '+' | '-' | '*' | '/' | ...
```



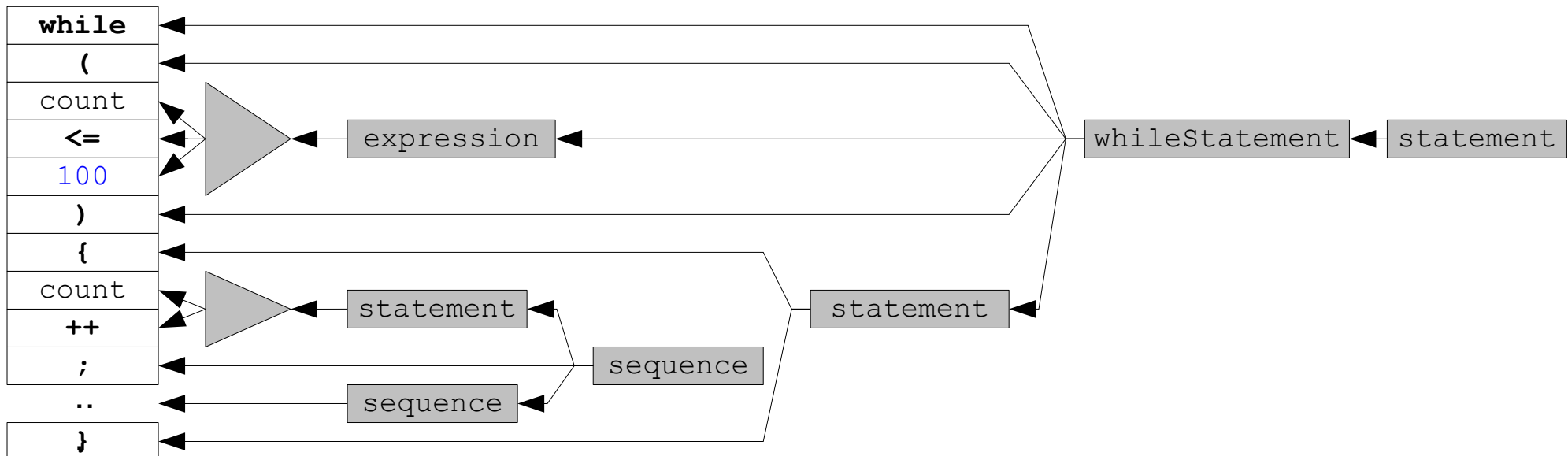
Assemblage et Compilation

- Exemple

Grammaire
du langage

```
statement: whileStatement
         | ifStatement
         | ...
         | '{' statementSequence '}'
whileStatement: 'while' '(' expression ')' statement
ifStatement: ...
sequence: ''
         | statement ';' sequence
expression: ...
```

Suite de
lexèmes



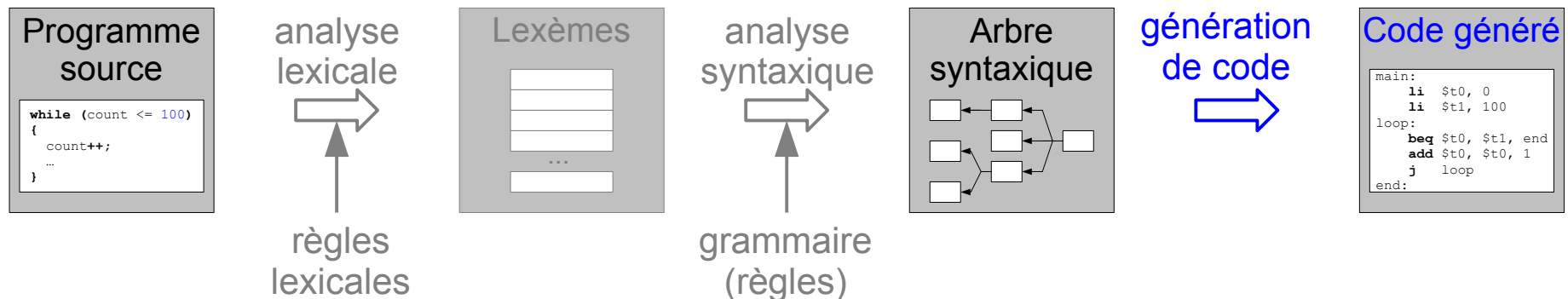
Tables des Matières

- Compilation
 - Analyse lexicale et syntaxique
 - Génération de code
- Assemblage
- Edition de liens

Assemblage et Compilation

- **Génération de code**

- Une fois la structure du programme source identifiée, le compilateur peut passer à la **génération de code**. Il s'agit de générer une suite d'instructions en langage d'assemblage qui corresponde à la structure du programme source.
 - Le compilateur doit identifier les opérations à réaliser et les instructions en langage d'assemblage qui y correspondent.
 - Le compilateur doit déterminer quels registres/emplacements mémoire doivent être utilisés pour stocker les variables et les résultats temporaires.



Assemblage et Compilation

- **Exemple – Expression arithmétique**

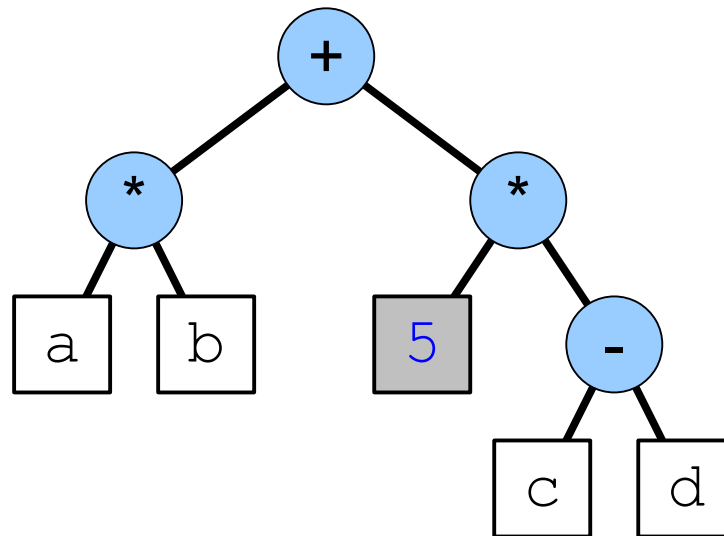
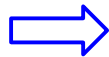
- Sur base de la priorité des opérateurs, telle que définie dans le langage, le sous-arbre syntaxique d'une expression arithmétique peut être converti en un arbre binaire représentant cette expression.

$a * b + 5 * (c - d)$

Suite de
lexèmes

a
*
b
+
5
*
(
c
-
d
)

analyse
syntaxique



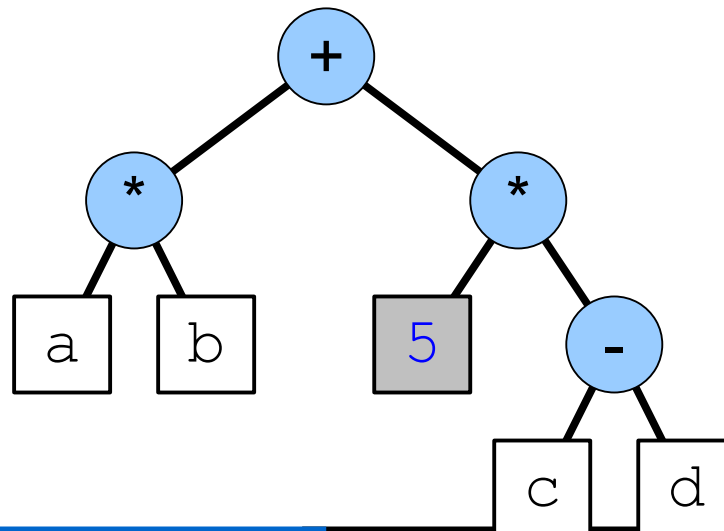
génération
de code



?

Assemblage et Compilation

- **Exemple – expression arithmétique**
 - Supposons que
 - 10 registres (t_0 à t_9) sont disponibles pour stocker des résultats temporaires.
 - les variables a , b , c et d sont des entiers 32 bits situés consécutivement aux adresses $0x10000000$ à $0x1000000C$.
 - le résultat de chaque opération arithmétique intermédiaire est placé dans un registre différent.

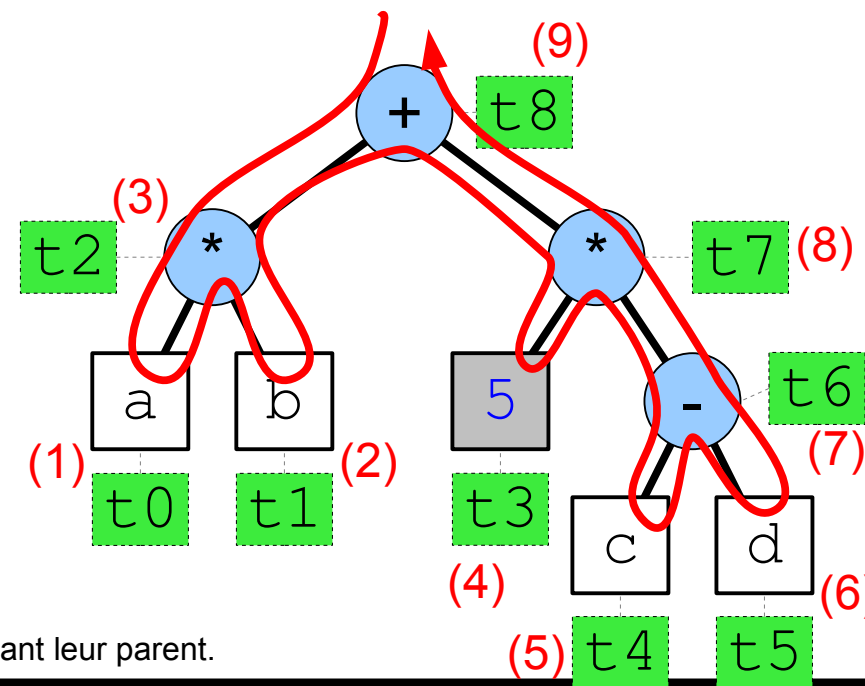


Assemblage et Compilation

- **Exemple – expression arithmétique**

- Un algorithme simple qui peut être utilisé par le compilateur pour générer une suite d'instructions correspondant à l'expression arithmétique consiste à parcourir l'arbre en *ordre postfixe* et à produire pour chaque nœud une suite d'instructions lui correspondant.

```
(1)  lw    $t0, 0x10000000    # a
(2)  lw    $t1, 0x10000004    # b
(3)  mult  $t0, $t1           # a*b
     mflo  $t2
(4)  li    $t3, 5             # 5
(5)  lw    $t4, 0x10000008    # c
(6)  lw    $t5, 0x1000000C    # d
(7)  sub   $t6, $t4, $t5      # c-d
(8)  mult  $t6, $t3           # 5*(c-d)
     mflo  $t7
(9)  add   $t8, $t2, $t7      # a*b+5*(c-d)
```



Note: lors d'un parcours postfixe, les sous-arbres de gauche et de droite sont traités avant leur parent.

Assemblage et Compilation

- **Exemple – expression arithmétique**
 - Le code généré utilise 11 instructions et 9 registres différents !
 - Il y a moyen de faire mieux. Une optimisation évidente consiste à ré-utiliser les registres qui contiennent un résultat temporaire dont on n'aura plus besoin par la suite.

```
lw    $t0, 0x10000000    # a
lw    $t1, 0x10000004    # b
mult  $t0, $t1           # a*b
mflo  $t2
li    $t3, 5             # 5
lw    $t4, 0x10000008    # c
lw    $t5, 0x1000000C    # d
sub   $t6, $t4, $t5      # c-d
mult  $t6, $t3           # 5*(c-d)
mflo  $t7
add   $t8, $t2, $t7      # a*b+5*(c-d)
```

Variables	Mémoire	Regs.
a	0x10000000	t0
b	0x10000004	t1
c	0x10000008	t4
d	0x1000000C	t5
temp. a*b		t2
temp. const. 5		t3
temp. c-d		t6
temp. 5*(c-d)		t7
temp. résultat		t8

Assemblage et Compilation

- **Exemple – expression arithmétique**
 - La version ci-dessous utilise la même suite d'instructions mais seulement 4 registres différents sont utilisés ($t0$ à $t3$).
 - Par exemple, après avoir effectué l'addition de a ($t0$) et b ($t1$), les valeurs de a et b ne sont plus nécessaires (il faut conserver leur addition en revanche) → il est possible de ré-utiliser les registres $t0$ et $t1$.

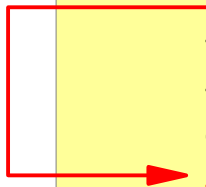
```
lw    $t0, 0x10000000    # a
lw    $t1, 0x10000004    # b
mult  $t0, $t1           # a*b
mflo  $t0
li    $t1, 5             # 5
lw    $t2, 0x10000008    # c
lw    $t3, 0x1000000C    # d
sub   $t2, $t2, $t3      # c-d
mult  $t1, $t2           # 5*(c-d)
mflo  $t1
add   $t0, $t0, $t1      # a*b+5*(c-d)
```


Assemblage et Compilation

- **Exemple – expression arithmétique**

- Voici encore une autre solution n'utilisant cette fois que **3 registres**. Les instructions sont identiques, mais une instruction a changé de place: il s'agit de l'instruction chargeant la constante 5.

```
lw    $t0, 0x10000000    # a
lw    $t1, 0x10000004    # b
mult  $t0, $t1           # a*b
mflo  $t0
lw    $t1, 0x10000008    # c
lw    $t2, 0x1000000C    # d
sub   $t1, $t1, $t2      # c-d
li    $t2, 5             # 5
mult  $t1, $t2           # 5*(c-d)
mflo  $t1
add   $t0, $t0, $t1      # a*b+5*(c-d)
```



Vérifiez par vous-même !...

Assemblage et Compilation

- **Optimisations**

- La génération de code est un problème difficile qui va au delà de l'objet de ce cours. Il existe beaucoup de solutions possibles pour l'allocation des registres et pour l'agencement des instructions mais il faut pouvoir trouver celle qui est la meilleure: il s'agit d'un problème d'optimisation.
- La définition de “*meilleur code*” varie en fonction des besoins. Par exemple, on privilégiera le code qui
 - utilise le moins d'espace en mémoire de programme
 - est le plus rapide
 - utilise le moins d'espace en mémoire de données
 - entraîne la consommation énergétique la plus faible
 - etc.

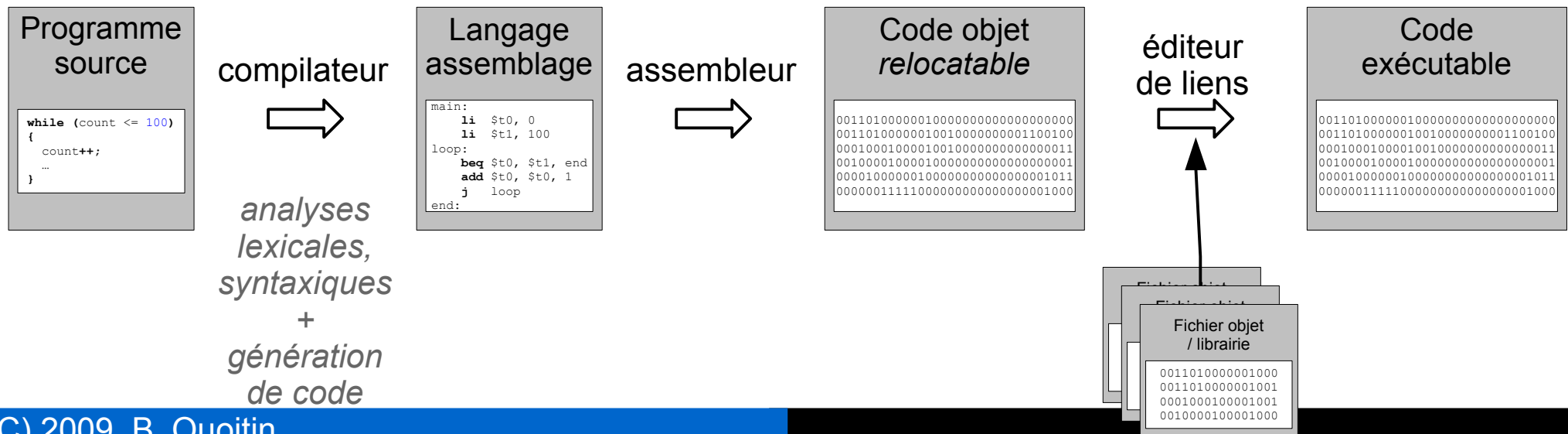
Tables des Matières

- Compilation
 - Analyse lexicale et syntaxique
 - Génération de code
- Assemblage
- Edition de liens

Assemblage et Compilation

- **Génération de code: chaîne de compilation**

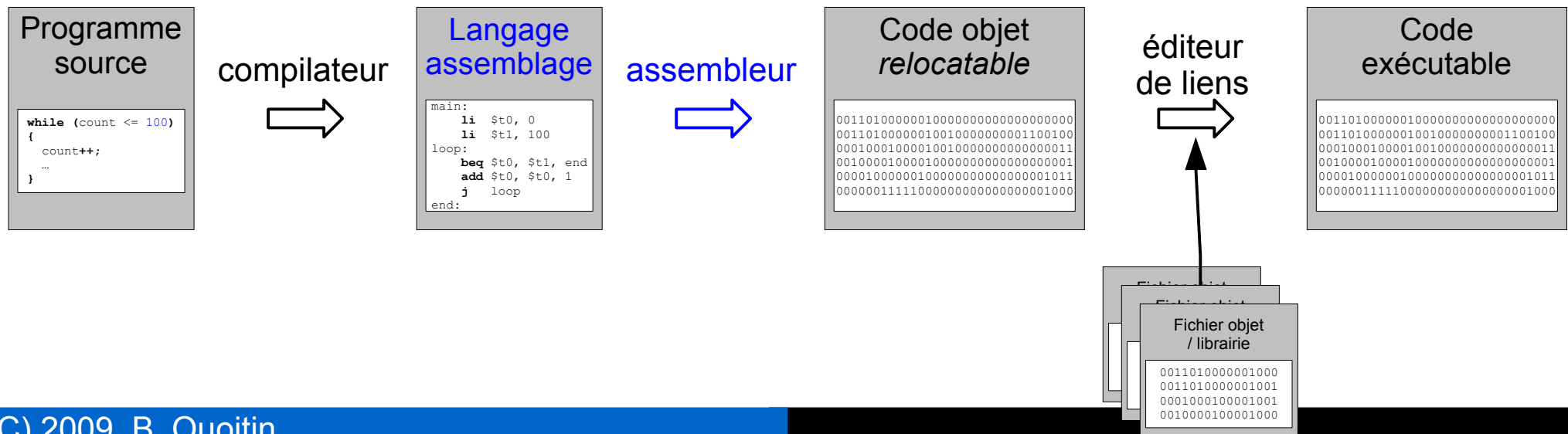
- Les compilateurs ne génèrent généralement pas directement du code exécutable. Plusieurs étapes intermédiaires ont lieu.
 - Le compilateur génère du code en langage d'assemblage.
 - L'**assembleur** convertit le programme du langage d'assemblage vers le langage machine et le place dans un **fichier objet** (*relocatable object file*).
 - L'**éditeur de liens** (*linker*) crée un exécutable à partir de plusieurs fichiers objets et/ou bibliothèques.



Assemblage et Compilation

- **Assembleur**

- L'assembleur a de multiples rôles
- **Traduire les instructions en langage d'assemblage** (mnémoniques + arguments) en instructions en langage machine, en respectant le format de l'architecture cible.
- **Traduire les symboles** tels que les noms des registres et les labels en adresses de registres et en adresses en mémoire.
- **Traduire les pseudo-instructions**



Assemblage et Compilation

- **Assembleur**

- Un assembleur fonctionne généralement en « deux passes ».

- **Première passe**

- Transformer les pseudo-instructions en instructions.
 - Attribuer à chaque instruction et donnée son adresse définitive dans le segment de code (`text`) ou de données (`data`).
 - Garder une table des symboles trouvés : **labels** et **noms de variables**.

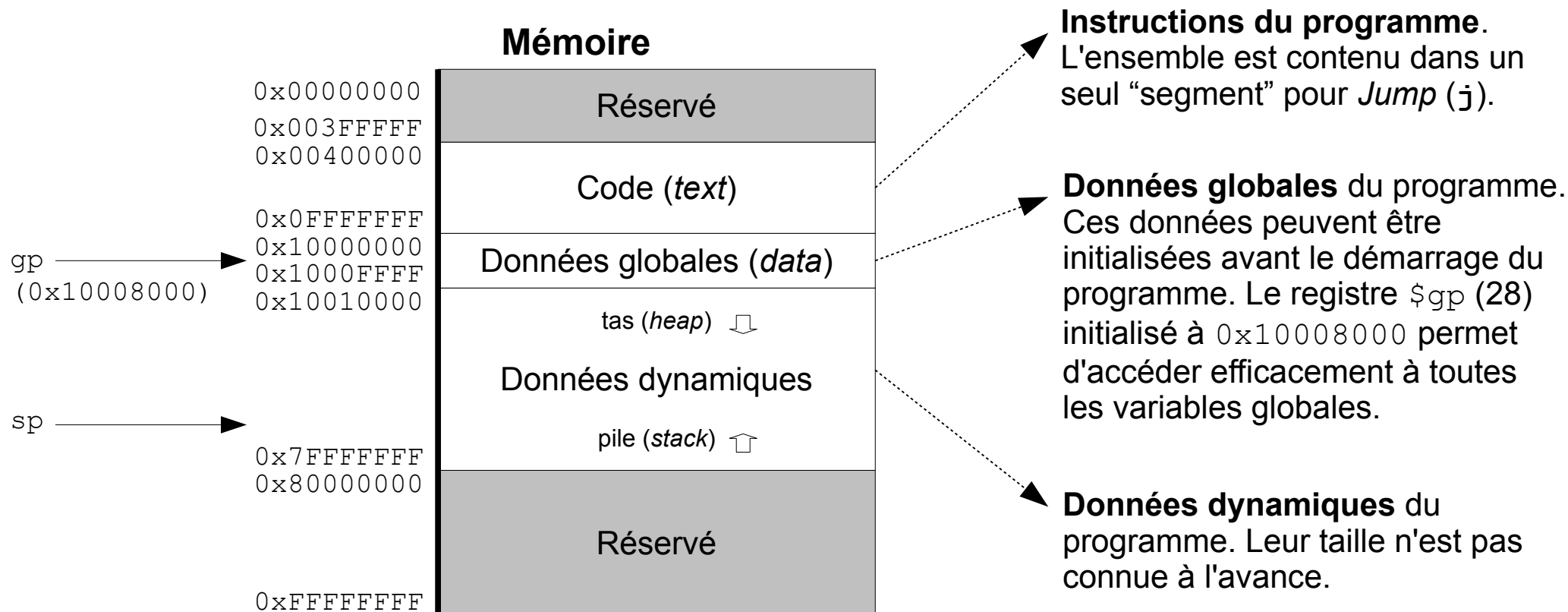
- **Deuxième passe**

- Générer le code machine. Les **adresses** correspondant aux variables et labels sont obtenues à partir de la table de symboles.
 - Le code généré et la table de symboles sont placés dans le fichier objet.

Assemblage et Compilation

- **Carte mémoire MIPS**

- La carte mémoire (*memory-map*) indique comment la mémoire est découpée. C'est l'ABI qui spécifie ce découpage.
- Exemple de l'ABI MIPS “o32”



Assemblage et Compilation

- **Exemple**

- Supposons que le programme en langage C ci-dessous soit traduit par le compilateur vers le programme en langage d'assemblage ci-contre.

Déclaration de mot de 32-bits en mémoire. Les labels permettent de les référencer.

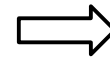
Variables globales

```
int f, g, h;

int main()
{
    f= 2;
    g= 3;
    h= sum(f, g);
    return h;
}

int sum(int a, int b)
{
    return a + b;
}
```

compilateur



```
.data
f:    .word 0
g:    .word 0
h:    .word 0

.text
main:
    addiu $sp, $sp, -4
    sw    $ra, 0($sp)
    li    $a0, 2
    sw    $a0, f
    li    $a1, 3
    sw    $a1, g
    jal   sum
    sw    $v0, h
    lw    $ra, 0($sp)
    addiu $sp, $sp, 4
    jr    $ra

sum:
    add   $v0, $a0, $a1
    jr    $ra
```

Note: exemple tiré de **Digital Design and Computer Architecture**, D. M. Harris and S. L. Harris, Morgan-Kaufmann, 2007.

Assemblage et Compilation

- **Assembleur: première passe**

- Lors de la première passe, chaque instruction et chaque donnée est placée à une **adresse** dans le segment de code (text) ou dans le segment de données (data). Une **table des symboles** est construite.

Adresses

```
0x00000000    main:
0x00000004      addiu $sp, $sp, -4
0x00000008      sw    $ra, 0($sp)
0x0000000C      li    $a0, 2
0x00000010      sw    $a0, f
0x00000014      li    $a1, 3
0x00000018      sw    $a1, g
0x0000001C      jal   sum
0x00000020      sw    $v0, h
0x00000024      lw    $ra, 0($sp)
0x00000028      addiu $sp, $sp, 4
0x0000002C      jr    $ra
0x00000030
0x0000002C    sum:
0x00000030      add   $v0, $a0, $a1
0x00000030      jr    $ra
```

```
0x00000000    f:      .word 0
0x00000004    g:      .word 0
0x00000008    h:      .word 0
```

Table des symboles

Symbole	Adresse	Segment
f	0x00000000	data
g	0x00000004	data
h	0x00000008	data
main	0x00000000	text
sum	0x0000002C	text

Assemblage et Compilation

- **Assembleur: deuxième passe**

- Lors de la deuxième passe, les **adresses ou offsets de branchement ou d'accès mémoire** sont calculés. Ensuite, le code des instructions est généré.

Adresses

```
0x00000000
0x00000004
0x00000008
0x0000000C
0x00000010
0x00000014
0x00000018
0x0000001C
0x00000020
0x00000024
0x00000028
0x0000002C
0x00000030
```

```
main:
    addiu $sp, $sp, -4
    sw    $ra, 0($sp)
    li    $a0, 2
    sw    $a0, f
    li    $a1, 3
    sw    $a1, g
    jal   sum
    sw    $v0, h
    lw    $ra, 0($sp)
    addiu $sp, $sp, 4
    jr    $ra

sum:
    add   $v0, $a0, $a1
    jr    $ra
```

```
0x00000000
0x00000004
0x00000008
```

```
f:    .word 0
g:    .word 0
h:    .word 0
```

Table des symboles

Symbole	Adresse	Segment
f	0x00000000	data
g	0x00000004	data
h	0x00000008	data
main	0x00000000	text
sum	0x0000002C	text

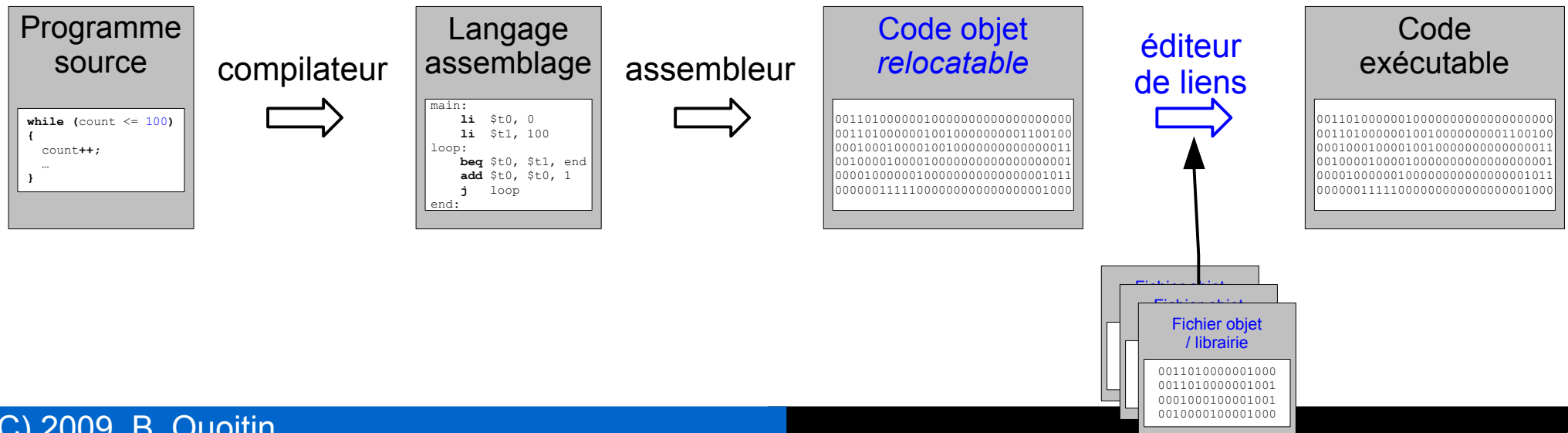
Tables des Matières

- Compilation
 - Analyse lexicale et syntaxique
 - Génération de code
- Assemblage
- Edition de liens

Assemblage et Compilation

- **Editeur de liens**

- Le rôle de l'**éditeur de liens** (*linker*) consiste à combiner plusieurs fichiers objets en un seul fichier appelé **fichier exécutable**.
 - Souvent un programme de grande taille est découpé en plusieurs fichiers objets afin de ne pas devoir tout recompiler lorsque des modifications d'une partie du code source sont effectuées.
 - Le *linker* va notamment devoir calculer les adresses de symboles externes aux différents objets, déplacer et compléter certaines instructions.

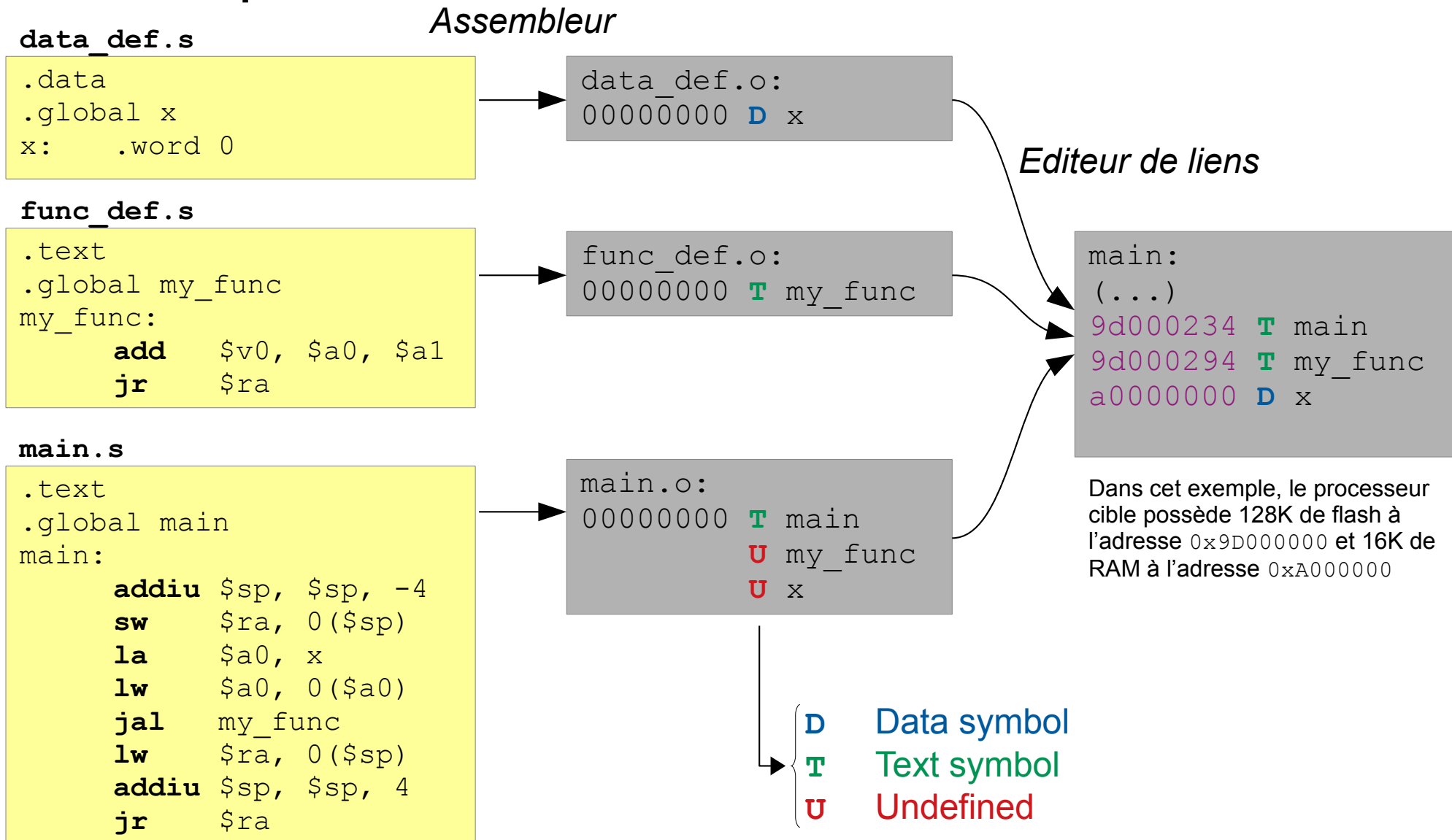


Assemblage et Compilation

- **Rôles de l'éditeur de liens**
 - **Lien avec les symboles externes**
 - Si un fichier objet contient un (ou plusieurs) symbole(s) indéfini(s), l'éditeur de liens **recherche ce(s) symboles dans les autres fichiers** objets. Si les symboles ne sont pas trouvés l'édition des liens échoue. Sinon, le symbole indéfini est remplacé par l'adresse du symbole trouvé.
 - **(Re-)positionnement du code**
 - Si plusieurs fichiers objets contiennent des instructions/données situées aux mêmes adresses, il faut les **repositionner** (*relocate*). Il faut aussi recalculer les adresses dans les instructions qui y font référence.
 - **Code d'initialisation**
 - Un **code d'initialisation** prédéfini peut éventuellement être adjoint (p.ex. *C RunTime initialization* - `crt`).

Assemblage et Compilation

• Exemple



Références

- **The Elements of Computing Systems: Building a Modern Computer From First Principles**, N. Nisan and S. Schocken, MIT Press, 2005.