UNIVERSITE PARIS-SUD

Master Informatique 1ere Année

Année 2016-2017

TER Compilation

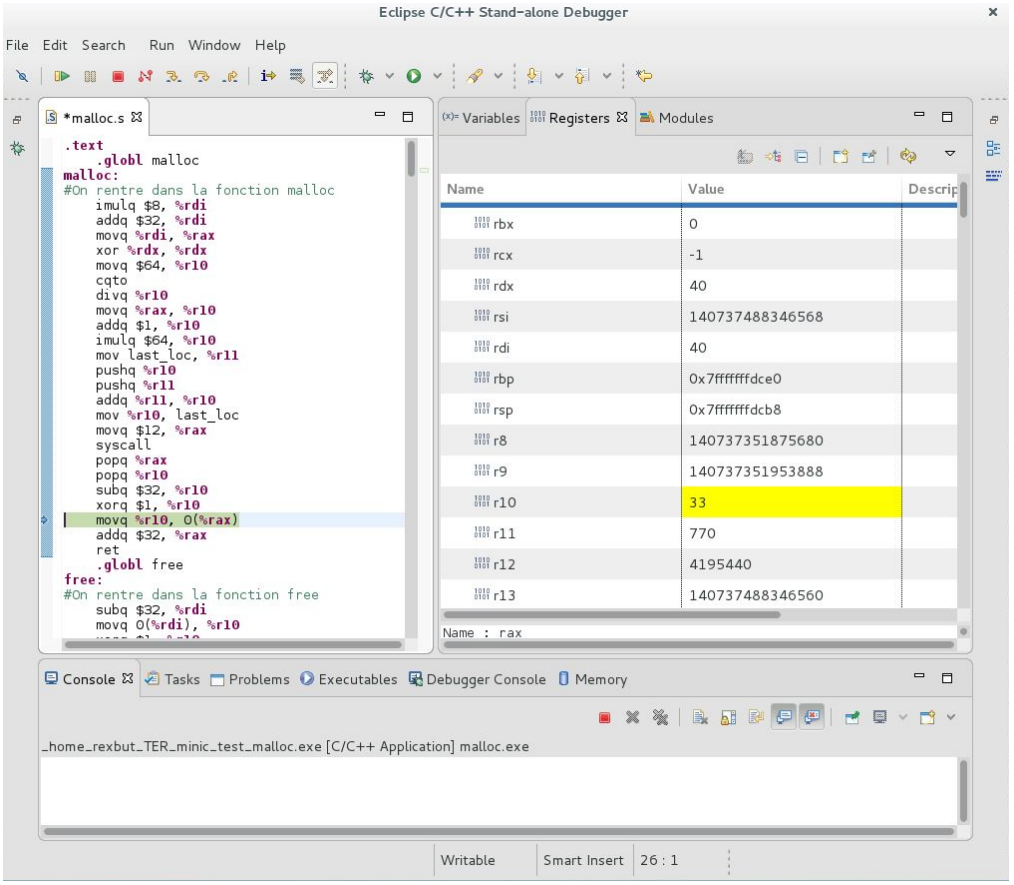
Rapport de Travaux d'Étude et de Recherche

par

Rémi BUTET
Dorian Graner

Enseignant : Kim.Nguyen, *Université Paris-Sud*

Sommaire



Introduction

Dans le cadre de l'unité d'enseignement TER Compilation du département Informatique de l'université Paris-Saclay, il a été demandé aux étudiants de Master 1, de réaliser en binôme l'implémentation d'un compilateur et une recherche bibliographique.

Notre compilateur ne doit compiler qu'un fragment du langage C, mais pour les codes C supportés, l'exécution d'un fichier compilé par notre compilateur doit être identique à celle d'un fichier compilé par un compilateur déjà existant comme gcc, en respectant la norme du langage. Notre projet couvre toutes les étapes de la compilation, de l'analyse lexicale à la production de code. Le langage de sortie est l'AMD64.

Nous avons choisi pour le sujet de recherche l'allocation mémoire car il nous semble intéressant de se pencher sur ce problème qui a une place importante dans un compilateur, et l'implémentation d'une version naïve de malloc et free dans notre compilateur nous parait un bon défi.

I. Conception du compilateur

1. L'organisation du programme

Le programme est composé de 8 fichiers :

- amd64.ml et amd64.mli, la bilbiothèque de génération de code AMD64 ;
- main.ml, le programme principal ;
- ast.mli, l'arbre de syntaxe du langage ;
- les 4 fichiers pour les étapes de compilation décrits ci-dessous.

a. Analyse lexicale (lexer.mll)

Le fichier dépend de Parser pour la liste des tokens et de Ast pour les types d'entiers.

L'analyse lexicale est la première étape de la compilation. Elle consiste à parcourir l'ensemble du fichier C pour le traduire en tokens utilisés pour les étapes suivantes. Les différents éléments sont décodés à l'aide d'expressions régulières :

- les chiffres entre 0 et 9 ;
- les entiers, une suite de chiffres ;
- les flottants, une suite de chiffres avec un point, ou avec la notation 'e-entier' pour les puissances de 10 ;
- le ident qui sont soit des identifiants, soit des mots clés du langage. La distinction est faite par la fonction keyword_or_ident ;
- les nombres en hexadécimal, un chiffre ou une lettre entre A et F ;
- les caractères, soit une lettre soit un caractère échappé (',",\,n,t ou x suivit d'un code hexa lui-même composé de deux nombres en hexa).

Le fichier C est parcouru caractère par caractère ou élément par élément, chacuns traduits à l'aide des fonctions adéquat et des expressions régulières ; ainsi un caractères entre quote est traduit en un entier correspondant au code du caractère calculé par la fonction decode_char par exemple. Les caractères du langage (opérateurs et parenthèses/crochets/accolades) ont chacun un token correspondant et la traduction est donc immédiate. Les espacements et commentaires sont ignorés puisqu'ils servent uniquement à la lecture du code mais aucunement à sa compilation.

Cet phase permet de détecter les erreurs d'écriture du programme C telles que les caractères et nombres mal formés et non reconnus ou bien les commentaires imbriqués (interdits par la consigne).

Exemple : en haut le code C, en bas la traduction en tokens

```
extern int printf() ;
int main () {
  printf("i \x3A %d\n", 24);
  return 0;
}
```

EXTERN INT IDENT(printf) LPAREN RPAREN SEMI *ligne suivante*
INT IDENT(main) LPAREN RPAREN LBRACE *ligne suivante*
IDENT(printf) LPAREN note1 COMA CONST_INT(Ast.Unsigned, Ast.Int, 24) RPAREN SEMI *ligne suivante*
RETURN CONST_INT(Ast.Unsigned, Ast.Int, 0) SEMI *ligne suivante*
RBRACE EOF

note1 : liste de caractères entre guillemets -> decode_char sur chaque caractère puis fusion des caractères décodés en une chaîne. La chaîne sera donc CONST_STRING (i : %d (retour à la ligne)).

b. Analyse syntaxique (parser.mly)

Le fichier dépend de Ast pour la liste des symboles.

L'analyse syntaxique consiste à associer la suite de tokens créés par l'analyse lexicale aux symboles de l'arbre de syntaxe abstraite. Dans ce fichier sont définis la liste des tokens existants (constantes, mot-clés, opérateurs, etc.) ainsi que l'associativité des opérateurs (droite si *a op b op c* signifie *a op (b op c)* ; gauche si *a op b op c* signifie *(a op b) op c*). Il contient ensuite une liste de règles pour faire l'association tokens/ast. Ainsi un fichier est composé d'une suite de déclarations de variables, structures et fonctions ; une déclaration de fonction contient une liste d'arguments entre des tokens LPAREN et RPAREN (parenthèses) séparés par un token COMA (virgule) ; un argument contient un type et une variable ; un type est soit le token VOID ou DOUBLE, soit le token STRUCT suivi d'un identifiant lui-même formé d'un token IDENT, etc. Grâce à toutes ces règles qui s'appellent entre elles, on obtient alors la structure du programme composé d'opérateurs unaire et binaires, de constantes, d'expressions, d'instructions, de déclarations, etc. Les erreurs syntaxiques sont détectées lorsqu'une suite de tokens ne correspond à aucune règle de la grammaire, comme souvent les parenthèses manquantes.

C'est aussi lors de cette étape que notre compilateur annote toutes ces expressions avec leur position (ligne et caractère) dans le programme à l'aide de la fonction mk_loc afin de pouvoir afficher cette information dans les messages d'erreur pour faciliter le débogage d'un programme qui ne compilerait pas.

Exemple : en haut le code C, en bas l'ast après analyse lexicale et syntaxique

<table>
<tr><td colspan="2">if(i==0) {
  i++;
}</td></tr>
<tr><td>Sif(
  Ebinop(Eq,Eident(i), Econst(0)),
  Sbloc(
    [],
    [Sexpr(Eunop(Post_dec, Eident(i)]
  ),
  None
)</td><td>Instruction if
Opérateur binaire ==, opérandes i et 0
Instruction bloc (“then”)
Liste vide des déclarations
Liste des instructions du bloc (une seule)
-> opérateur unaire --, opérande i
Instruction bloc(“else”) vide

(pour des raisons de clarté, les informations de localisation ne sont pas représentées.</td></tr>
</table>

c. Typage (typing.ml)

Le fichier dépend de Ast pour la liste des symboles.

Le typage est la partie la plus ardue de la compilation puisqu'elle consiste à vérifier la cohérence de type du programme en entier afin de s'assurer qu'il pourra être traduit et exécuté. Le fichier contient des environnements afin de stocker les variables, structures et fonctions typées, des fonctions auxiliaires pour l'affichage d'erreurs et la comparaison de types, et les fonctions de typages. Ces dernières consistent à annoter les différentes expressions et instructions avec leur type et s'appeler mutuellement afin de s'assurer la cohérence des types. La fonction type_expr s'occupe des expressions, en s'assurant par exemple que tous les opérateurs binaires sont utilisés avec deux opérandes compatibles, ce qui implique de typer chaque opérande en tenant compte des cast possibles, des pointeurs etc., ou encore qu'un appelle de fonction concerne bien une fonction déclarée préalablement et que le type des arguments attendus est compatible avec ceux donnés. La fonction type_lvalue concerne les expressions utilisées comme valeurs gauche et est donc appelée par la précédente lors du typage de certaines expressions. Les instructions sont typées par type_instr, c'est-à-dire notamment le if et le for pour les plus intéressantes. La fonction type_decl est utilisée comme son nom l'indique pour les déclaration, en s'assurant que l'objet déclaré n'existe pas déjà et le cas échéant l'ajoute à l'environnement adéquat via la fonction auxiliaire add_env ; pour les fonctions la présence d'une instruction return de bon type est vérifiée. Enfin, type_prog sert de point d'entrée en lançant la vérification récursive et vérifie aussi qu'une fonction “main”, obligatoire en C, bien formée est présente.

Toutes les erreurs de typage sont détectées et affichées avec les données précises de localisation, d'identifiants et l'erreur précise rencontrée pour simplifier encore une fois le débogage.

Exemple :

double* f;
extern int printf();
int fact(int n) {
  if (n <= 1) return 1;
  return n * fact(n - 1);
}
 	
int main () {
  int i;
  i = fact(4);
  printf("i \x3A %d\n",i);
  printf("%d\n", &f);
  for(i=0; i<3; i++){
  f = f+i;
  printf("%d\n", f);
  }
  return 0;
}
	Le pointeur sur double f est ajouté à l'environnement des variables globales.
	L'extern printf est ajoutée à l'environnement des fonctions.
	La fonction fact est ajoutée à l'environnement des fonctions. L'entier n est ajouté à l'environnement local de la fonction. Le corps de la fonction est typé (les différentes instructions et expressions) en s'assurant que toutes les branches d'exécutions ont un return du bon type (ici int).
	La fonction main est ajoutée à l'environnement des fonctions. L'entier i est ajouté à l'environnement local de la fonction. Le résultat de fact est bien un entier, sa valeur est stockée à l'adresse de i sur la pile. Les arguments passés aux appels de printf sont de types compatibles donc le typage des appels est cohérent. Les expressions en argument de la boucle for sont typées, i est un entier donc le typage est valide. Le bloc de la boucle for est typé, l'addition f+i est entre un entier et un pointeur donc valide, il s'agit bien d'un décalage de 8 en 8 puisque c'est un incrémentation d'une adresse. Le return est bien de type entier.
	L'entièreté du programme est bien typé et il y a une fonction main valide, le typage est correct.

d. Production de code (compile.ml)

Le fichier dépend de AMD64 pour la production de code AMD64 et de Typing pour les affichages d'erreurs.

Le typage ayant assuré la cohérence du code C, la dernière étape consiste à traduire ce code vers l'AMD64. Le code de sortie est donc une liste d'instructions assembleur ainsi qu'un champ data contenant le constantes du programme. La traduction consiste à appeler récursivement les différentes fonctions spécifiques (instructions, expressions, etc.) sur le main, le fichier a donc une structure assez similaire à celui du typage. On trouve deux environnements pour stocker respectivement les chaînes de caractères et les constantes flottantes qui iront dans le segment data du code AMD64, des fonctions auxiliaires (principalement pour des tailles de registres ou de décalages, ainsi que pour la création de labels pour les instructions et fonctions), et les fonctions non plus de typage mais de traduction. Ainsi, compile_cast permet l'ajustement de la taille des registres lors du cast d'un type vers un autre de taille différente ; compil_const créé les labels pour les constantes et les ajoutent à l'environnement correspondant ; compile_expr_reg stock le résultat d'une instruction dans un registre en gérant les différents cas selon la taille des opérandes des opérateurs, les flottants, etc. ; compile_block traduit les blocs instructions en utilisant un label où revenir lorsque les instructions sont exécutées ainsi qu'un offset pour retrouver la position des variables locales sur la pile ; compile_decl gère les déclarations, et notamment les fonctions qui doivent déplacer les pointeurs de pile, appeler compile_block, stocker le retour si il y en a, puis nettoyer la pile une fois leur exécution terminée afin de ne pas la polluer (la fonction main est traitée légèrement différemment puisque son retour est dans le registre de retour du programme) ; enfin compile_prog lance la récursion et écrit le corps du programme dans le segment text du fichier et les constantes dans data.

Le corps du programme est en fait une longue chaîne de caractères d'où l'utilisation de l'opérateur ++ pour écrire chaque instruction assembleur à la suite et le compilateur n'a plus qu'à écrire le segment tel quel dans le corps du programme assembleur de sortie.
Exemple : en haut le code C, en bas le code assembleur

```
double* f;
extern int printf();
int fact(int n) {
  if (n <= 1) return 1;
  return n * fact(n - 1);
}
int main () {
    int i;
    i = fact(4);
    printf("i \x3A %d\n",i);
    printf("%d\n", &f);
    for(i=0; i<3; i++){
                f = f+i;
                printf("%d\n", f);
    }
    return 0;
}
```

```
.text
   .globl fact
fact:
#On rentre dans la fonction fact
   pushq %rbp
   mov %rsp, %rbp
   addq $-8, %rsp
   movl $1, %r10d
   andq $-1, %r10
   pushq %r10
   leaq 16(%rbp), %r10
   mov 0(%r10), %r10d
   popq %r11
   cmp %r11d, %r10d
   setle %r10b
   movzbl %r10b, %r10d
   test %r10, %r10
   je __label__else_00001
   movl $1, %r10d
   andq $-1, %r10
   pushq %r10
   jmp fact_fin
   jmp __label__end_00002
__label__else_00001:
__label__end_00002:
   subq $8, %rsp
   movl $1, %r10d
   andq $-1, %r10
   pushq %r10
   leaq 16(%rbp), %r10
   mov 0(%r10), %r10d
   popq %r11
   sub %r11d, %r10d
   andq $-1, %r10
   pushq %r10
   call fact
   addq $8, %rsp
   popq %r10
   andq $-1, %r10
   pushq %r10
   leaq 16(%rbp), %r10
   mov 0(%r10), %r10d
   popq %r11
   imul %r11d, %r10d
   andq $-1, %r10
   pushq %r10
   jmp fact_fin
fact_fin:
   popq %r10
   mov %r10, 24(%rbp)
   mov %rbp, %rsp
   popq %rbp
   ret
   .globl main
main:
#On rentre dans la fonction main
   pushq %rbp
   mov %rsp, %rbp
   addq $-16, %rsp
   subq $8, %rsp
   movl $4, %r10d
   andq $-1, %r10
   pushq %r10
   call fact
   addq $8, %rsp
   popq %r10
   andq $-1, %r10
   pushq %r10
   leaq -8(%rbp), %r10
   popq %r11
   mov %r11d, 0(%r10)
   movq %r11, %r10
   andq $-1, %r10
   pushq %r10
   popq %r10
mov $__label__string_00003, %r10
   pushq %r10
   popq %rdi
   leaq -8(%rbp), %r10
   mov 0(%r10), %r10d
   andq $-1, %r10
   pushq %r10
   popq %rsi
   pushq %rsp
   pushq 0(%rsp)
   andq $-16, %rsp
   movq $0, %rax
   call printf
   movq 8(%rsp), %rsp
   mov %rax, %r10
   andq $-1, %r10
   pushq %r10
   popq %r10
   mov $__label__string_00004,
%r10
   pushq %r10
   popq %rdi
   movq $f, %r10
   pushq %r10
   popq %rsi
   pushq %rsp
   pushq 0(%rsp)
   andq $-16, %rsp
   movq $0, %rax
   call printf
   movq 8(%rsp), %rsp
   mov %rax, %r10
   andq $-1, %r10
   andq $-1, %r10
   pushq %r10
   leaq -8(%rbp), %r10
   popq %r11
   mov %r11d, 0(%r10)
   movq %r11, %r10
   andq $-1, %r10
   pushq %r10
   popq %r10
__label__if_00005:
   movl $3, %r10d
   andq $-1, %r10
   pushq %r10
   leaq -8(%rbp), %r10
   mov 0(%r10), %r10d
   popq %r11
   cmp %r11d, %r10d
   setl %r10b
   movzbl %r10b, %r10d
   test %r10, %r10
   je __label__end_00006
   leaq -8(%rbp), %r10
   mov 0(%r10), %r10d
   andq $-1, %r10
   pushq %r10
   movq $f, %r10
   mov 0(%r10), %r10
   popq %r11
   mov $8, %r12
   imul %r12, %r11
   add %r11, %r10
   pushq %r10
   movq $f, %r10
   popq %r11
   mov %r11, 0(%r10)
   movq %r11, %r10
   pushq %r10
   popq %r10
   mov $__label__string_00004,
%r10
   pushq %r10
   popq %rdi
   movq $f, %r10
   mov 0(%r10), %r10
   pushq %r10
   popq %rsi
   pushq %rsp
   pushq 0(%rsp)
   andq $-16, %rsp
   movq $0, %rax
   call printf
   movq 8(%rsp), %rsp
   mov %rax, %r10
   andq $-1, %r10
   pushq %r10
   popq %r10
   leaq -8(%rbp), %r10
   mov 0(%r10), %r11d
   mov %r11d, %r12d
   inc %r11d
   mov %r11d, 0(%r10)
   mov %r12d, %r10d
   andq $-1, %r10
   pushq %r10
   popq %r10
   jmp __label__if_00005
__label__end_00006:
   movl $0, %r10d
   pushq %r10
   jmp main_fin
main_fin:
   popq %rax
   mov %rbp, %rsp
   popq %rbp
   ret
.data
f:
   .align 8
   .space 8
__label__string_00003:
   .string "i : %d\n"
__label__string_00004:
   .string "%d\n"
```

II. Allocateur de mémoire

L'allocation de mémoire peut se faire de différentes façons selon le moment et la méthode utilisés. La première différence se situe entre l'allocation statique au moment du chargement du programme ou allocation dynamique durant son déroulement. L'allocation statique se déroule donc au chargement du programme en mémoire et concerne l'espace mémoire dont on connaît la taille avant l'exécution, l'information est alors formellement ajoutée dans l'exécutable à la compilation et l'OS réserve l'espace nécessaire avant de commencer l'exécution. L'allocation dynamique quant à elle se fait durant le déroulé du programme – par exemple pour un tableau dont la taille est calculée à l'exécution – et peut être faite soit sur la pile soit sur le tas. La pile est gérée automatiquement par le programme pour stocker les variables et les instructions des fonctions, tandis que le tas sert aux allocations par le programmeur avec malloc et free.

1. Mémoire virtuelle et paging

a. Principe général

L'objectif de la mémoire virtuelle est de pallier les limites de la mémoire réelle en permettant à un programme (ou plusieurs exécutés en même temps) d'utiliser plus de mémoire que ce qui est réellement disponible en s'appuyant sur le fait que la totalité de la mémoire nécessaire au(x) programme(s) n'est pas utilisée en même temps. La mémoire virtuelle donne au programme l'illusion d'avoir accès à plus de mémoire que c'est le cas en mettant à disposition des adresses mémoire virtuelles qui sont ensuite traduites en adresses réelles (ou "physiques") sans que le programme ne s'en rende compte. De cette façon, plusieurs adresses virtuelles peuvent en fait mener à la même adresse réelle à laquelle seront chargées des informations différentes selon le programme ou la partie de programme qui y accède.

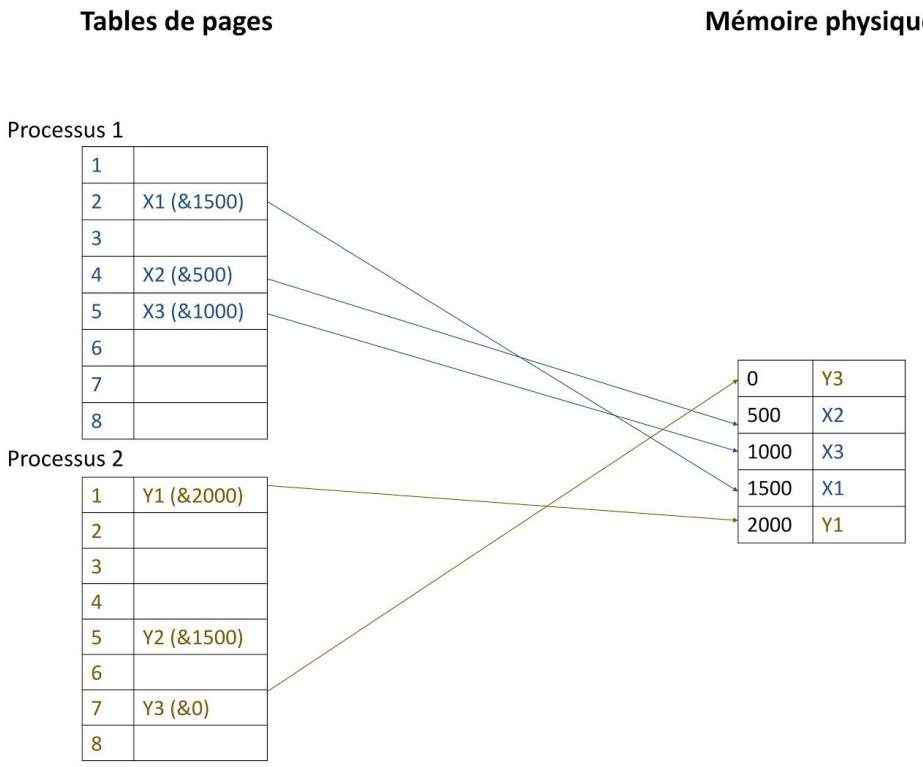
b. Pages et cadres

La mémoire virtuelle est divisée en pages toutes de même taille auxquelles les programmes ont l'illusion d'accéder. La mémoire physique est, elle, divisée en cadres de pages, qui délimitent des zones où sont stockées les données. Chaque programme possède sa propre table des pages qui fait la correspondance entre page et cadre de page, c'est ce mécanisme qui permet à plusieurs pages de mémoire virtuelle d'en fait correspondre à la même adresse physique. C'est le MMU (Memory Management Unit) qui s'occupe de la traduction des adresses via la table des pages. Ainsi, quand un programme croit accéder à une adresse, il accède en fait à l'adresse physique résultant de la traduction de cette adresse grâce à sa table de page par le MMU.

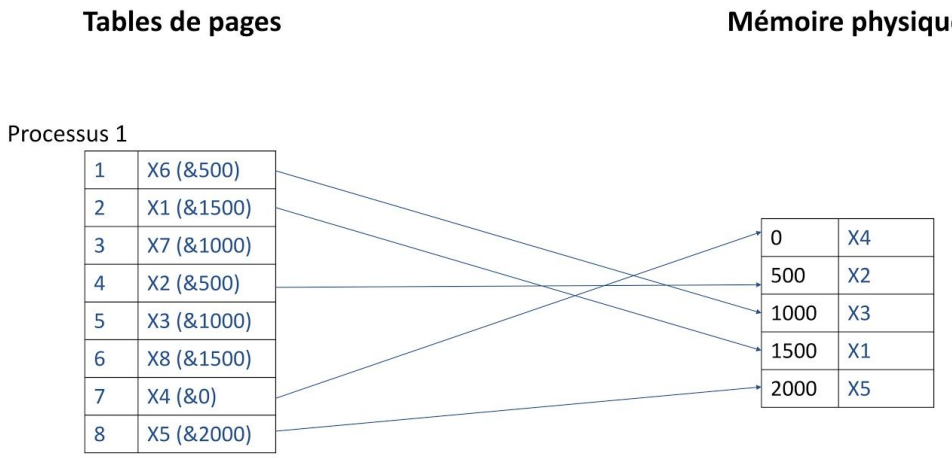
Cette structure possède de nombreux avantages, elle facilite notamment l'édition de lien puisque le programme croit accéder à une adresse fixe, et c'est l'adresse physique

correspondante qui varie, ainsi que l'allocation de mémoire puisque le programme peut demander une grande quantité de pages contiguës (un gros tableau de structures par exemple) en mémoire virtuelle mais qui ne seront pas forcément contiguës en mémoire physique.

Le fait que plusieurs pages puissent pointer vers une même adresse physique et qu'une quantité de mémoire virtuelle bien supérieur à la mémoire physique puisse être allouée entraîne cependant aussi des difficultés techniques puisqu'il faut choisir quand remplacer une page par une autre dans le cadre de page sur la mémoire physique, optimiser ces changements de pages, par exemple lorsque plusieurs programmes tournent en parallèle et que par conséquent des pages de ces différents programmes sont chargées en même temps. Ce problème peut se traduire par une forte lenteur si de nombreux programmes se partagent une faible mémoire physique, multipliant les défauts de page (accès à une page pas encore chargée en mémoire physique) puisque chacun a de bonnes chances de remplacer les pages du programme précédent par le sienne, ce qui est coûteux, puisqu'il n'y a plus de place vide en mémoire physique.



Si le processus 1 veut accéder à l'adresse 4 pour récupérer la valeur X2, le MMU va traduire cette adresse en l'adresse 500 en mémoire physique où X2 est réellement stockée. Les deux processus ont une page qui pointe à l'adresse 1500, donc si le processus 2 essaye d'accéder à Y2, il y aura un défaut de page et l'OS va devoir la charger à la place de X1 dans la mémoire physique.



Le processus 1 a l'illusion d'avoir accès à 8 blocs mémoire alors qu'en réalité seulement 5 sont disponibles en mémoire physique. Ainsi, les adresses 1 et 4 en mémoire virtuelles pointent à la même adresse physique, l'OS devra donc échanger les pages dans le cadre de page à l'adresse 500 si le programme accède à X2 puis X6.

2. Implémentation de *malloc* et *free*

Les fonctions *malloc* et *free* permettent respectivement de réserver et de libérer dynamiquement un bloc de mémoire. Les blocs mémoires alloués sont stockés dans le tas, il est donc indispensable de la libérer par la suite, sinon cela provoque une fuite mémoire.

```
void *malloc(int size);
```

La signature de la fonction malloc

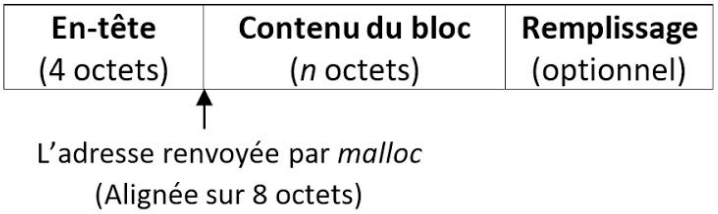
La fonction *malloc* prend comme seul paramètre le nombre d'octets à allouer et retourne un pointeur vers un nouveau bloc d'au moins *size* octets. Cependant en cas échec, typiquement il n'y a plus de mémoire disponible, la fonction renverra NULL.

```
void free(void *ptr);
```

La signature de la fonction free

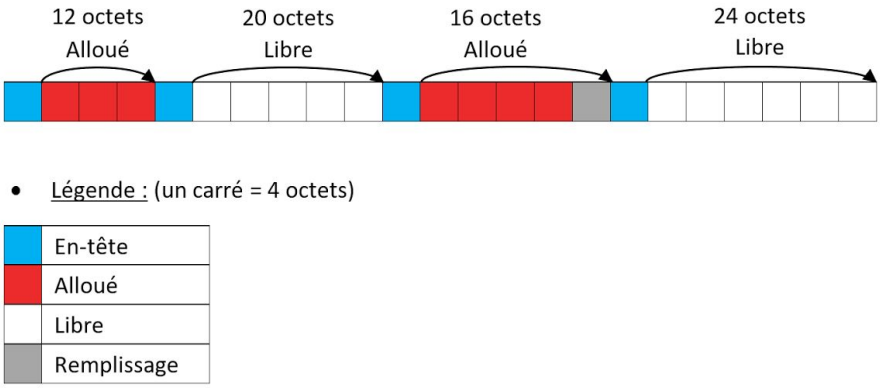
La fonction *free* prend comme argument l'adresse du premier octet du bloc alloué et ne retourne aucune valeur. Cependant si le pointeur n'a pas été alloué par la fonction malloc, il peut se produire une erreur.

a. Mise en œuvre



Structure d'une zone mémoire alloué (sans optimisations)

Dans le tas les blocs alloués et libres sont contigus, donc pour pouvoir réaliser une gestion correcte de la mémoire. Il a fallu y ajouter un entête qui contient la taille et le statut du bloc. La taille correspond au nombre d'octets du bloc et le statut permet de savoir s'il est alloué ou libre. De plus pour des raisons d'optimisation de performances lors de l'accès mémoire, l'adresse renvoyée par malloc est alignée sur 8 octets. C'est pour cela qu'il peut y avoir des valeurs de remplissage dans la mémoire.



Organisation de la mémoire (sans optimisations)

Comme expliqué précédemment l'en-tête contient la taille et le statut du bloc. Grâce au fait que la taille des zones mémoires soit des multiples de 8, il est possible d'utiliser les trois bits de poids faible pour sauvegarder le statut :

bit	5	4	3	2	1	0	Taille	Statut
…	0	1	0	0	0	1	Taille 16	Alloué
…	0	1	0	0	0	0	Taille 16	Libre
…	0	1	1	0	0	1	Taille 24	Alloué
…	1	0	0	0	0	0	Taille 32	Libre

Exemple de valeur d'en-tête

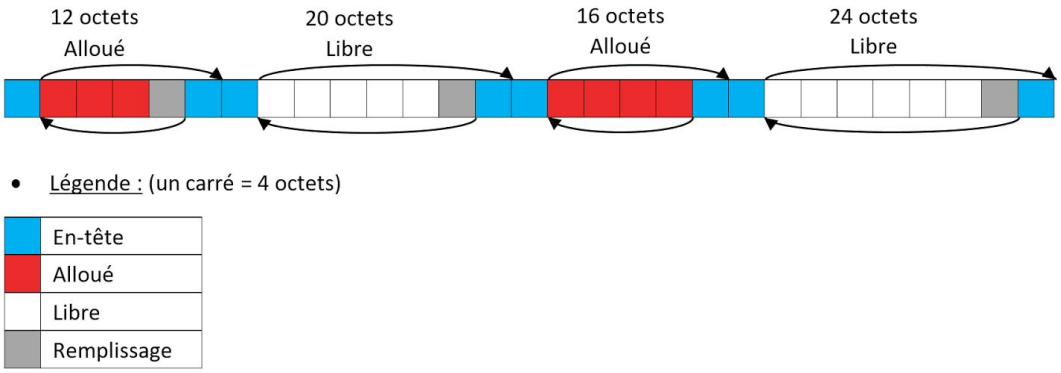
Lorsque la fonction malloc est appelé, il cherche un bloc mémoire libre suffisamment assez grand. S'il en trouve un alors il le découpe éventuellement en deux (bloc alloué + bloc libre), puis il renvoie le bloc alloué. Sinon il alloue un nouveau bloc à la fin du TAS et le renvoie. Pour choisir quel bloc prendre, il existe plusieurs stratégies qui ont chacun leurs avantages mais aucune réellement mieux que les autres :

- First fit : Le premier bloc assez grand
- Next fit : Le premier bloc assez grand mais en commençant la recherche où s'était arrêté la précédente
- Best fit : Un bloc assez grand de taille minimale

Cependant au bout d'un moment, à force de diviser les blocs en 2 parties la mémoire sera fragmentée. Cela rendra inutilisable une grande partie de la mémoire et augmentera le coût de recherche.

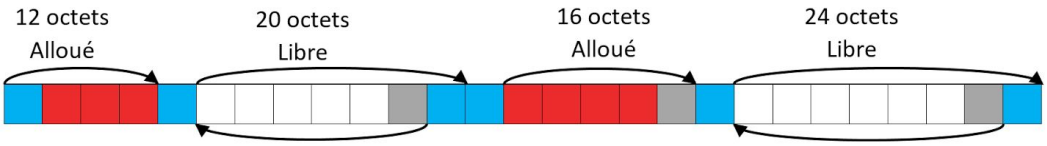
b. Fusion des blocs libres

Lorsqu'un bloc est libéré au lieu de changer uniquement le statut, il faut aussi déterminer s'il peut être fusionné avec les blocs libres adjacents. Ce qui permet de regrouper tous les blocs libres adjacents et récupéré un grand bloc libre. Cela est simple de vérifier si le bloc suivant est libre car chaque bloc dispose d'assez d'information pour aller au bloc suivant. Cependant il ne dispose d'aucune information sur le bloc précédent, il existe une solution simple qui consiste à dupliquer l'en-tête à la fin d'un bloc. Cette solution vient de Donald Knuth et est appelée *"boundary tags"*.



Organisation de la mémoire (boundary tags)

Cette méthode est assez coûteuse en mémoire mais il existe quelques améliorations qui permettent de réduire cela. Par exemple, il est possible de ne dupliquer les en-têtes uniquement sur les blocs libres et de sauvegarder directement dans l'en-tête si le bloc précédent est libre ou pas.



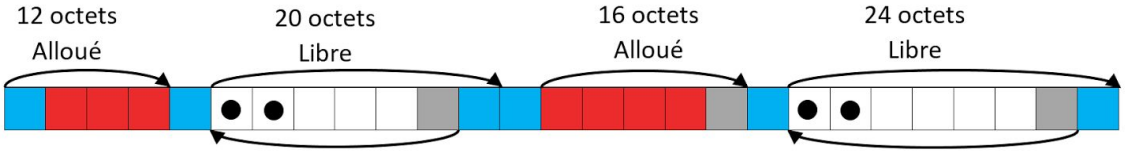
- Légende : (un carré = 4 octets)

	En-tête
	Alloué
	Libre
	Remplissage

Organisation de la mémoire (boundary tags et optimisation)

c. Free List

Parcourir toute la mémoire pour trouver des blocs libres assez grands reste quand même assez coûteux. C'est de là que vient l'idée de chaîner les blocs libres entre eux, ce qui permet de se déplacer directement au bloc libre suivant. Pour cela il suffit d'utiliser le contenu du bloc qui est actuellement vide, en y ajoutant 2 pointeurs. Mais le grand inconvénient c'est qu'il faut une taille minimale par bloc.



- Légende : (un carré = 4 octets)

	En-tête
	Alloué
	Libre
●	Libre (avec pointeur)
	Remplissage

Organisation de la mémoire (free list)

Cela permet donc de naviguer beaucoup plus rapidement entre les blocs libres et de gérer simplement l'ordre d'utilisation des blocs libres :

- Privilégier les blocs libres les plus anciens
- Privilégier les blocs libres les plus récent
- Trier les blocs libres par adresses croissantes
- Trier les blocs libres par taille
- Créer une liste par taille de bloc

Il y a énormément d'autre possibilité d'implémentations et c'est ce qui rend chaque implémentation d'allocation mémoire unique. Actuellement, il y a environ 5000 lignes dans le fichier *malloc.c* de Linux et il est en constante évolution. Cependant il faut savoir faire un compromis entre l'optimisation de temps de calcule et la quantité mémoire utilisée pour le faire fonctionner.

Conclusion

Lors de la réalisation de ce projet nous avons rencontré plusieurs difficultés lors des différentes étapes du compilateur, notamment la gestion des chaînes et commentaires lors de l'analyse lexicale ainsi la gestion des pointeurs par l'analyse syntaxique – comportement différent avec opérateurs, etc. – et par le typage – toutes les opérations sur pointeur. Cependant, le déroulé du projet consistant à écrire tout le processus de compilation avec l'aide du professeur nous a permis de bien comprendre la structure d'un compilateur, l'utilité et le comportement de chacune des phases sans pour autant rester bloquer par nous-même face à la complexité de certaines parties. Il est de plus très intéressant de voir peu à peu s'étoffer les fichiers de test au fur et à mesure que le compilateur se complexifie, de passer de la simple détection d'identifiants redéfinis à la compilation d'un programme plus complexe comme Mandelbrot. La partie recherche nous a permis d'approfondir nos connaissances sur l'allocation mémoire en général mais particulièrement sur malloc/free, tout en nous obligeant à réaliser un travail de synthèse et en réfléchissant à comment l'appliquer dans notre propre compilateur. Il s'agit donc d'un projet très intéressant par lui-même et par le fait qu'il est dans la continuité directe du cours de compilation du premier semestre.

La partie implémentation de l'allocation et free a été finalement la plus grosse difficulté rencontrée, car la traduction de notre algorithme en AMD64 a posé d'importants problèmes. La version implémentée ne couvre donc pas l'intégralité des fonctionnalités de malloc prévues initialement.

Références bibliographiques

- Malloc : https://fr.wikipedia.org/wiki/Malloc
- Compilation et langages, par Thibaut Balabonski (2016) : https://www.lri.fr/~blsk/Compilation/Compil5-Memoire.pdf
- Memory allocation : https://courses.cs.washington.edu/courses/cse351/10sp/lectures/15-memallocation.pdf
- Free list : https://en.wikipedia.org/wiki/Free_list
- Gestion de la mémoire en C : http://ilay.org/yann/articles/mem/
- Mécanismes d'allocation dynamique : http://rperrot.developpez.com/articles/c/allocationC/
- La pagination, la segmentation et la mémoire virtuelle : http://www-etud.iro.umontreal.ca/~gottif/bdeb/infc32/c8.htm
- Linux Memory Allocation and Forcing Memory Release : http://heapspray.net/post/linux-memory-allocation-and-forcing-memory-release/
- Ouvrage : "*Le langage C*" par Brian W. Kernighan et Dennis M. Ritchie
- Ouvrage : "*Computer Systems : A Programmer's Perspective*" par Randal E. Bryant et David R. O'Hallaron