Rappels de programmation

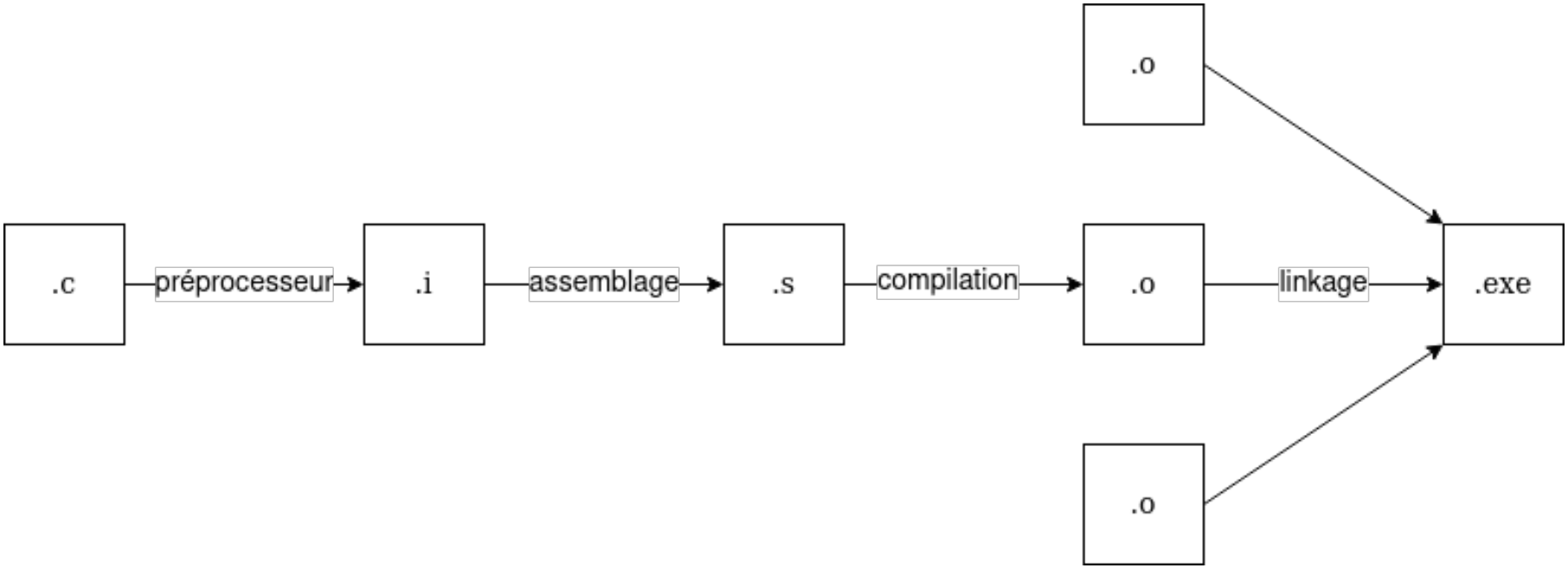
Quentin Fortier

September 1, 2024

Compilation

Passage d'un code source à un exécutable :

1. Préprocesseur : inclusion des fichiers d'en-tête, remplacement des macros, suppression des commentaires.
2. Compilation : traduction du code source en assembleur.
3. Édition de liens : regroupement des différents fichiers objets en un exécutable.



Compilation

En pratique :

- `gcc fichier.c` : compile `fichier.c` en `a.out`.
- `gcc -o prog fichier.c` : compile `fichier.c` en `prog`.
- `gcc fichier1.c fichier2.c` : compile `fichier1.c` et `fichier2.c` en `a.out`.
- `gcc *.c` : compile tous les fichiers `.c` en `a.out`.
- `gcc -Wall ...` : affiche tous les warnings et les erreurs.
- `gcc -fsanitize=address ...` : détecte les erreurs de mémoire.
- `gcc -lm ...` : pour utiliser les fonctions mathématiques (`math.h`).

Trouver les erreurs

Exercice

- Trouver les erreurs dans les programmes suivants.
- Les corriger.

Liste

```
let somme l =
    let s = 0 in
    for i = 0 to List.length l - 1 do
        s := s + l.(i)
    done;
    s
```

Liste

```
let somme l =
    let s = 0 in
    for i = 0 to List.length l - 1 do
        s := s + l.(i)
    done;
    s
```

- `t.(i)` existe seulement pour un tableau, pas une liste.
- Impossible de modifier une variable en OCaml à moins que ce soit une référence.

Liste

```
let somme l =
    let s = 0 in
    for i = 0 to List.length l - 1 do
        s := s + l.(i)
    done;
    s
```

```
let rec somme l = match l with
    | [] -> 0
    | t::q -> t + somme q
```

Liste

```
let range n =
    let l = [] in
    for i = 0 to n - 1 do
        i::l
    done;
    l
```

Liste

```
let range n =
    let l = [] in
    for i = 0 to n - 1 do
        i::l
    done;
    l
```

- `i::l` ne modifie pas `l`, mais renvoie une nouvelle liste.
- Il n'y a pas de append en OCaml.
- Une « liste » en Python est en fait un tableau.

Liste

```
let range n =
    let l = [] in
    for i = 0 to n - 1 do
        i::l
    done;
    l
```

```
let rec range n =
    if n = -1 then []
    else (n - 1)::(range (n - 1))
```

return

```
let appartient t e =
    for i = 0 to Array.length t - 1 do
        if t.(i) = e then true
    done;
    false
```

return

```
let appartient t e =
    for i = 0 to Array.length t - 1 do
        if t.(i) = e then true
    done;
    false
```

Pas de `return` en OCaml.

return

```
let appartient t e =
    for i = 0 to Array.length t - 1 do
        if t.(i) = e then true
    done;
    false
```

```
let appartient t e =
    let r = ref false in
    for i = 0 to Array.length t - 1 do
        if t.(i) = e then r := true
    done;
    !r
```

match

```
let appartient l e = match l with
    | [] -> false
    | e::q -> true
    | t::q -> appartient q e
```

match

```
let appartient l e = match l with
    | [] -> false
    | e::q -> true
    | t::q -> appartient q e
```

e est une nouvelle variable dans le **match**, qui écrase le e en argument.

match

```
let appartient l e = match l with
    | [] -> false
    | e::q -> true
    | t::q -> appartient q e
```

```
let appartient e l = match l with
    | [] -> false
    | t::q -> if t = e then true
        else appartient e q
```

Passage d'arguments

```
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

Passage d'arguments

```
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

Un appel `swap(x, y)` se fait par **copie des arguments** : `a` et `b` sont des copies de `x` et `y` (qui ne sont pas modifiées).

Passage d'arguments

Passage d'argument par valeur/copie

```
void f(int x) {
    x = 42;
}

int y = 0;
f(y); // y n'est pas modifié
```

```
def f(x):
    x = 42

y = 0
f(y) # y n'est pas modifié
```

Passage d'arguments

Passage d'argument par valeur/copie

```
void f(int x) {
    x = 42;
}

int y = 0;
f(y); // y n'est pas modifié
```

```
def f(x):
    x = 42

y = 0
f(y) # y n'est pas modifié
```

Passage d'argument par adresse/référence

```
void f(int* x) {
    *x = 42;
}

int y = 0;
f(&y); // y est modifié
```

```
let f t =
    t.(0) <- 42

let y = [|0|] in
f y (* y est modifié *)
```

Passage d'arguments

```
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

Passage d'arguments

```
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

```
void swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Variable locale

```
float* zero(){
    float t[2];
    t[0] = t[1] = 0;
    return t;
}
```

Variable locale

```
float* zero(){
    float t[2];
    t[0] = t[1] = 0;
    return t;
}
```

```
float* zero(){
    float* t = {0, 0};
    return t;
}
```

Variable locale

```
float* zero(){
    float t[2];
    t[0] = t[1] = 0;
    return t;
}
```

```
float* zero(){
    float* t = {0, 0};
    return t;
}
```

`t` est une variable locale qui est détruite à la fin de la fonction.

Variable locale

```
float* zero(){
    float* t = {0, 0};
    return t;
}
```

```
float* zero(){
    float* t = (float*)malloc(2 * sizeof(float));
    t[0] = t[1] = 0;
    return t;
}
```