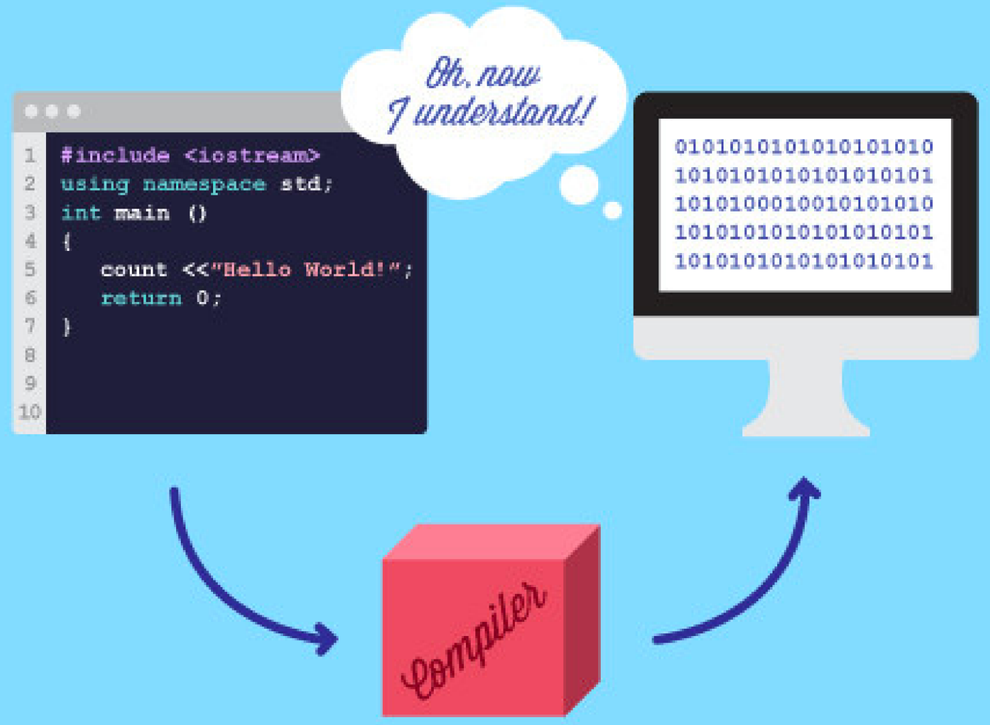
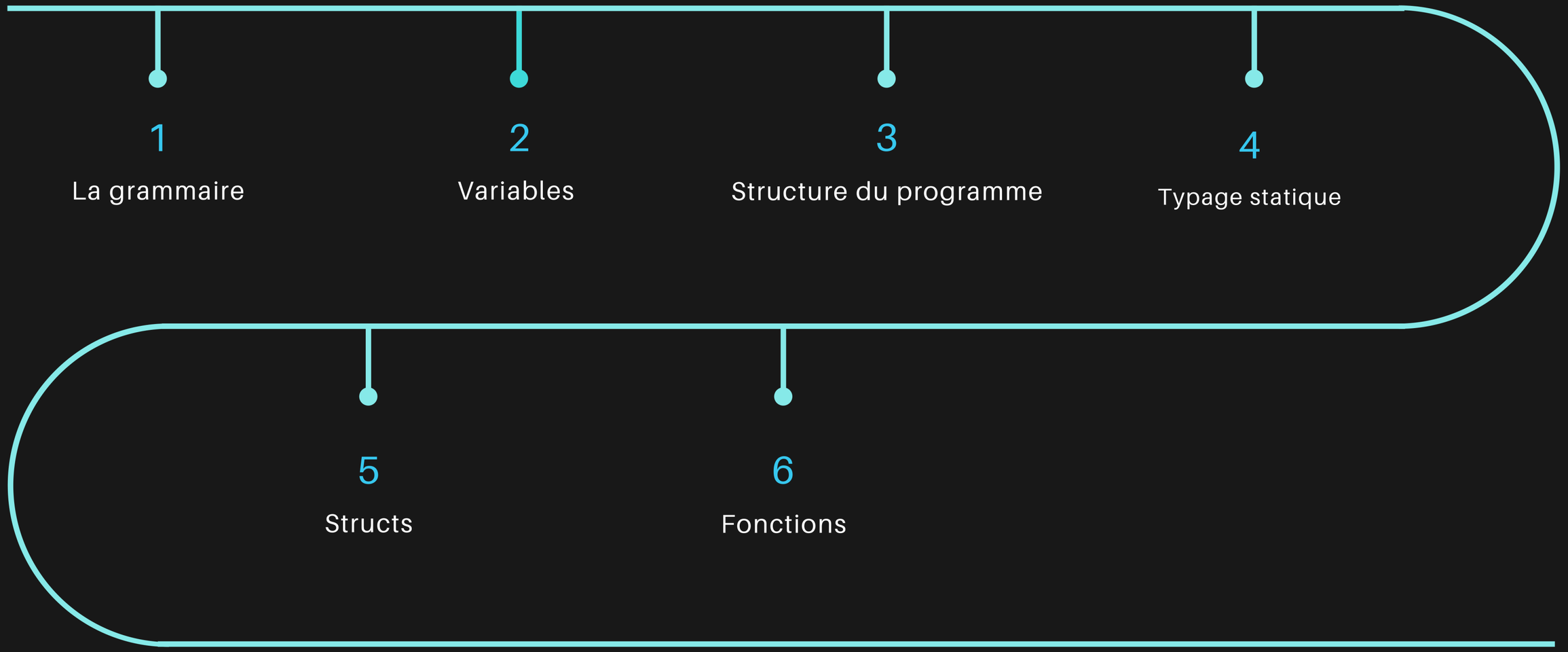
Advanced Compilation :

- Typage Statique
- Structures
- Fonctions

Jhon Sebastian ROJAS RODRIGUEZ
RHANEM Fatine
ENFROY Émilie



Sommaire :


1
La grammaire
2
Variables
3
Structure du programme
4
Typage statique
5
Structs
6
Fonctions

La grammaire :

```
exp : SIGNED_INT                    -> exp_int
| SIGNED_FLOAT                      -> exp_float
| "'" CHAR "'"                      -> exp_char
| IDENTIFIER                        -> exp_var
| IDENTIFIER "." IDENTIFIER         -> exp_var_struct
| exp OPBIN exp                     -> exp_opbin
| "(" exp ")"                       -> exp_par
| function_call                     -> exp_function

com : dec                                   -> declaration
| IDENTIFIER "=" exp ";"                    -> assignation
| IDENTIFIER "." IDENTIFIER "=" exp ";"     -> assignation_struct_var
| "if" "(" exp ")" "{" bcom "}"             -> if
| "while" "(" exp ")" "{" bcom "}"          -> while
| "print" "(" exp ")" ";"                   -> print
| function_call ";"                         -> function_call
```

```
function_call : IDENTIFIER "(" exp_list ")"

bdec : (dec)+

bcom : (com)*

dec : TYPE IDENTIFIER ";" -> declaration
| "struct" IDENTIFIER IDENTIFIER ";" -> declaration_struct
| TYPE IDENTIFIER "=" exp ";" -> declaration_expression
| "struct" IDENTIFIER IDENTIFIER "=" exp ";" -> declaration_struct_expression

struct : "struct" IDENTIFIER "{" bdec "}" ";"

function : TYPE IDENTIFIER "(" var_list ")" "{" bcom "return" exp ";" "}"        -> function_return
| "void" IDENTIFIER "(" var_list ")" "{" bcom "}"                                -> function_void
| "struct" IDENTIFIER IDENTIFIER "(" var_list ")" "{" bcom  "return" exp ";" "}" -> function_return_struct

bstruct : (struct)*

bfunction : (function)*

prg : bstruct bfunction "int" "main" "(" var_list ")" "{" bcom "return"  exp ";"  "}"
```

```
exp_list: -> vide
| exp ("," exp)*  -> at_least_one_expression

var_list :                          -> vide
| ((TYPE IDENTIFIER)|(IDENTIFIER IDENTIFIER))("," ((TYPE IDENTIFIER)|(IDENTIFIER IDENTIFIER)))*  -> at_least_one_variable

IDENTIFIER : /[a-zA-Z][a-zA-Z0-9]*/
CHAR : /./
TYPE : "int" | "double" | "float" | "char" | "long"

OPBIN : /[+\-]/

%import common.WS
%import common.SIGNED_NUMBER
%import common.SIGNED_INT
%import common.SIGNED_FLOAT
%ignore WS
""",start="prg")
```

Variables :

```
section .bss
ans32 : resd 1
ans64 : resq 1

book.bookId : resd 1
book.title : resb 1
book.rating : resd 1
book.critRating : resq 1
book.profit : resq 1
getNewBook.bookId : resd 1
getNewBook.title : resb 1
getNewBook.rating : resd 1
getNewBook.critRating : resq 1
getNewBook.profit : resq 1
```

```
section .data
fmti : db "%i", 10, 0
fmtli : db "%li", 10, 0
fmtlf : db "%lf", 10, 0
fmtf : db "%f", 10, 0
fmtc : db "%c", 10, 0
argc : dq 0
argv : dq 0
const0_32: dd 5.0
const0_64: dq 5.0
const1_32: dd 4.5
const1_64: dq 4.5
const2_32: dd 2.0
const2_64: dq 2.0
const3_32: dd 4.8
const3_64: dq 4.8
const4_32: dd 3.8
const4_64: dq 3.8
```

```
""" Dict to store all declared variables
{
    "variable_name" : "variable_type",
    ...
}
"""
variables = {}
```

```
const_count = 0 #to keep track of the index of each floating point constant found
"""Dict to store the floating point constants found in the code
{
    "const" : "const_index",
    ...
}
"""
constants = {}
```

On a fait des fonctions qui traversent l'arbre en remplissant les dictionnaires

```
""" Dict to store the parameters of each declared function and its return type
{
    "function_name" : {
        "parameters" :{
            "parameter_name" : "parameter_type",
            ...
        },
        "return": "return_type"
        ...
    },
    ...
}
"""
functions = {}


 """ Dictionary to store member's information for each declared struct
 {
     "struct_name": {
         "member_name" : "member_type",
         ...
     },
     ...
 }
 """
 structs = {}
```

Structure du programme

```python
def asm_prg(p):
    f = open("template.asm")
    moule = f.read()
    vars_prg(p)
    C = asm_bcom(p.children[3])
    moule = moule.replace("BODY", C)
    E = asm_exp(p.children[4])
    moule = moule.replace("RETURN", E)
    F = asm_bfunction(p.children[1])
    moule = moule.replace("FUNCTIONS", F)
    D = asm_decl_vars()
    moule = moule.replace("DECL_VARS", D)
    constants_asm = asm_decl_const()
    moule = moule.replace("DECL_CONSTS", constants_asm)
    s = ""
    for i in range(len(p.children[2].children)//2):
        v = p.children[2].children[2*i+1].value
        e = f"""
        mov rbx, [argv]
        mov rdi, [rbx + { 8*(i+1)}]
        xor rax, rax
        call atoi
        mov [{v}], rax
"""
        s = s + e
    moule = moule.replace("INIT_VARS", s)
    return moule
```

```
1  extern printf
2  extern  atoi
3  global main
4  section .data
5
6  fmti : db "%i", 10, 0
7  fmtli : db "%li", 10, 0
8  fmtlf : db "%lf", 10, 0
9  fmtf : db "%f", 10, 0
10 fmtc : db "%c", 10, 0
11 argc : dq 0
12 argv : dq 0
13 DECL_CONSTS
14
15 section .bss
16 ans32 : resd 1
17 ans64 : resq 1
18
19 DECL_VARS
20
21 section .text
22
23 FUNCTIONS
24
25 main :
26     push rbp
27     mov [argc], rdi
28     mov [argv ], rsi
29 INIT_VARS
30 BODY
31 RETURN
32     mov rdi , fmti
33     mov rsi , rax
34     call printf
35     pop rbp
36     ret
```

Typage Statique:

Déclaration de variables: On déclare les variables comme globales dans la partie .bss du fichier asm avec la taille correspondante à chaque type (resb, resd et resq ou db, dd et dq pour les constants)

Inférence de types: On a construit une fonction pour rechercher automatiquement les types associés à des expressions.

```
exp : SIGNED_INT                    -> exp_int
| SIGNED_FLOAT                      -> exp_float
| "'" CHAR "'"                      -> exp_char
| IDENTIFIER                        -> exp_var
| IDENTIFIER "." IDENTIFIER         -> exp_var_struct
| exp OPBIN exp                     -> exp_opbin
| "(" exp ")"                       -> exp_par
| function_call                     -> exp_function
```

Typage Statique:

Opérations: Pour les opérations impliquant des expressions de nombres à virgule flottante, il est nécessaire d'utiliser l'Unité de calcul en virgule flottante (FPU).

```
book1.critRating = f - 3;

mov rax, 3
mov [ans32], rax
fild dword [ans32]
mov rax, [f]
mov [ans32], rax
fld dword [ans32]
fsub st1
fst dword [ans32]
fstp qword [ans64]
mov rax, [ans64]
mov [book1.critRating], rax
```

Fonction Print: On a besoin de savoir quel format on doit utiliser pour afficher le résultat d'une expression donc on utilise l'inférence de types pour assigner le format correspondant avant appeler la fonction printf.

```
mov rax, [book1.rating]
mov [ans32], rax
mov rdi, fmtf
fld dword [ans32]
fstp qword [ans64]
movq xmm0, qword [ans64]
mov rax, 1
call printf
```

Structs:

Normalement le compilateur gère les structs en utilisant la pile, il détermine la taille que va occuper chaque struct et il fixe l'offset de chaque membre. On gère les structs à l'aide d'un dictionnaire qui garde la structure de chaque struct et on utilise les variables déclarées directement.

```
struct Book{
    int    bookId;
    char   title;
    float  rating;
    double critRating;
    long   profit;
};
```

```
newBook.bookId : resd 1
newBook.title : resb 1
newBook.rating : resd 1
newBook.critRating : resq 1
newBook.profit : resq 1
```

```
""" Dictionary to store member's information for each declared struct
{
    "struct_name": {
        "member_name" : "member_type",
        ...
    },
    ...
}
"""
structs = {}
```

```
mov rax, [book2.bookId]
mov [book1.bookId], rax
mov rax, [book2.title]
mov [book1.title], rax
mov rax, [book2.rating]
mov [ans32], rax
mov rax, [ans32]
mov [book1.rating], rax
mov rax, [book2.critRating]
mov [ans64], rax
mov rax, [ans64]
mov [book1.critRating], rax
mov rax, [book2.profit]
mov [book1.profit], rax
```

Les Fonctions :

- Les variables des fonctions sont **globales** dans le code assembleur définies comme pour le main, dans DEC VAR.
- Lorsqu'on écrit x= test(x,y); dans le main du C :

```
def asm_com(c):
    elif e.data == "exp_function":
        return asm_function_call(e.children[0])
```

Fonction_call :

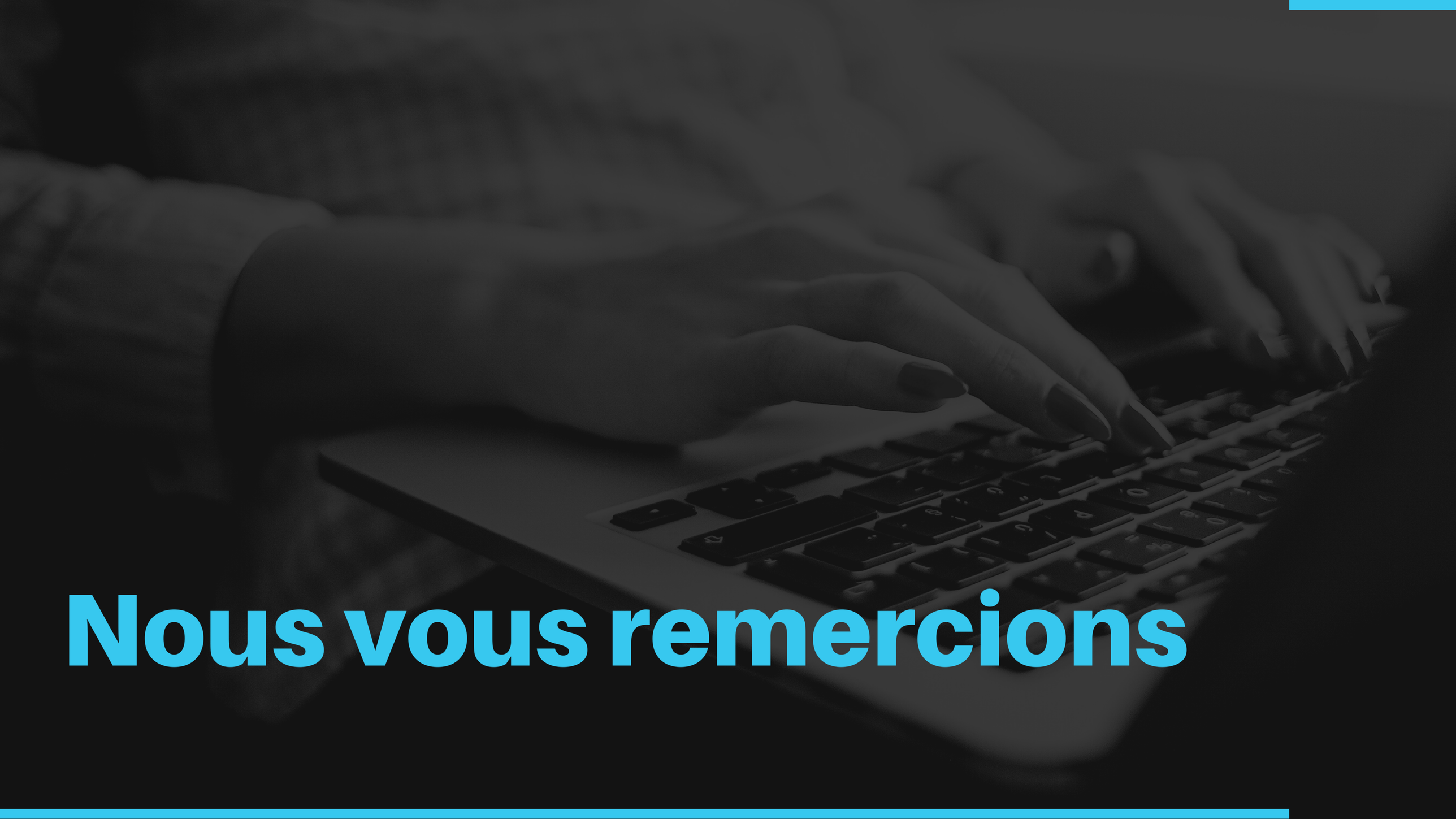
```python
def asm_function_call(fc):
    function_name = fc.children[0].value
    n_parameters = len(functions[function_name]['parameters'])
    exp_list = fc.children[1].children

    if len(exp_list) != n_parameters:
        raise Exception(f"error: wrong number of parameters to call {function_name}")
    asm = ""
    for i in range(n_parameters):
        parameter_name = list(functions[function_name]['parameters'].keys())[i]
        parameter_type = list(functions[function_name]['parameters'].values())[i]
        expression = exp_list[i]
        if parameter_type in basic_types:
            asm += asm_assignation(parameter_name, expression)
        else:
            verify_struct_expression(parameter_type, expression)
            if expression.data == "exp_function":
                asm += asm_function_call(expression.children[0])
            right_struct = get_struct_var_name_from_expression(expression)
            asm += asm_assign_struct(parameter_name, right_struct, parameter_type)
    asm += f"call {function_name}\n"
    return asm
```

Assignation :

```
def asm_assignation(var_name, expression):
    var_type = variables[var_name]
    types = type_exp(expression)
    expression = get_no_par_expression(expression)
    if var_type not in basic_types: #struct
        verify_struct_expression(var_type, expression)
        asm = ""
        if expression.data == "exp_function":
            asm += asm_function_call(expression.children[0])
        right_struct = get_struct_var_name_from_expression(expression)
        asm += asm_assign_struct(var_name, right_struct, var_type)
        return asm
    asm = asm_exp(expression)
    if "double" in types or "float" in types:
        if var_type not in types:
            raise Exception(f"error: invalid type in assignation")
        asm += f"\tmov rax, [ans{32 if var_type == 'float' else 64}]\n"
    asm += f"\tmov [{var_name}], rax\n"
    return asm
```

Les Fonctions :

```
def asm_function(f):
    if f.data == "function_void":
        name = f.children[0]
        command_block=asm_bcom(f.children[2])
        s=f"""
{name}:
    push rbp
    mov rbp,rsp
    """
        s=s+f"""
        {command_block}
        leave
        ret
"""
        return s
    elif f.data =="function_return":
        TYPE =f.children[0]
        name=f.children[1]
        command_block=asm_bcom(f.children[3])
        EXP=asm_exp(f.children[4])
        s=f"""
{name}:
    push rbp
    mov rbp,rsp
    {command_block}
    """
        s=s+EXP
        s=s+f"""
    leave
    ret
        return(s)
    elif f.data == "function_return_struct":
        struct_name = f.children[0].value
        function_name = f.children[1]
        bcom = asm_bcom(f.children[3])
        exp = f.children[4]
        verify_struct_expression(struct_name, exp)
        asm_return = ""
        if exp.data == "exp_var":
            asm_return = asm_assign_struct(function_name, exp.children[0].value, struct_name)
        elif exp.data == "exp_function":
            asm_return = asm_assign_struct(function_name, exp.children[0].children[0], struct_name)
        return (
    f"""
{function_name}:
    push rbp
    mov rbp,rsp
    {bcom}
    {asm_return}
    leave
    ret
""")
```



Nous vous remercions