Univ. Grenoble Alpes **UFR IM2AG**
MoSIG1 **Academic Year 2016 - 2017**

Programming Languages and Compiler Design

Final Exam - Wednesday December, 7 2016

Guidelines and information:

- Duration: 3 hours.
- Documents from the lecture and tutorial are authorised. Electronic devices are forbidden.
- The grading scale is indicative.

Part 1: Typing ($\sim$ 4 points)

In this part, we are interested in fragments of language **While** allowing to declare programs without variable declarations which would allow to decide the type of variable. The type of a variable can only be computed during an assignment. We consider a fragment of language **While** seen in the course :

$$\begin{array}{lll} e & ::= & n \mid x \mid e \odot e \mid \cdots \\ S & ::= & x := e \mid S; S \mid \texttt{if}\ e\ \texttt{then}\ S\ \texttt{else}\ S\ \texttt{fi} \mid \texttt{while}\ e\ \texttt{do}\ S\ \texttt{od} \end{array}$$

For expressions, the binary operator $\odot$ denotes any arithmetical or logical operator. Concerning statements S, we consider assignment, conditional, sequential, and iterative statements:

- The assignment of the value of an expression to a variable. The variable keeps the last value which it has been assigned to and thus its type. When a variable is assigned several times in a program, the system keeps only its last value and thus the type of the last value. For example,

  ```
  x := 3;
  x := x=4
  ```

  In this sequence, type `Int` is associated to `x` and then type `Bool`.

- A conditional statement. The two parts of a conditional assignment should conduct to the definition of the same variables with coherent types for each branch.

  Hence, if variables `a` and `b` were previously assigned, statement `if a=b then x:= 4 else y:=5` will be incorrect; as well as `if a=b then x := 4 else x := 4=6`. However, statement `if a=b then (x:=4;y:=3) else (y:=1;x:=8)` is correct.

Computing types We consider the sets:

- Names which contains the set of variable names,
- Types = {`Int`, `Bool`},
- Env = Names $\rightarrow$ Types $\ni \Gamma$, Γ is a function which associates a type to each variable name.

We consider the following judgments for statements `S` and expressions `e`:

- for statements: $\Gamma \vdash S \mid \Gamma'$, which means that in the environment Γ, statement S is correctly typed and produces environment Γ'.
- for expressions: $\Gamma \vdash e : t$, which means that in environment Γ, expression e is well typed and of type t.

$\dfrac{\Gamma(x) = \texttt{Int}}{\Gamma \vdash x : \texttt{Int}}$	x is of type `Int` in environment Γ if $\Gamma(x) = \texttt{Int}$.
$\dfrac{}{\Gamma \vdash n : \texttt{Int}}$	denotation n is of type `Int`.
$\dfrac{\Gamma \vdash e_1 : \texttt{Int} \quad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash e_1 + e_2 : \texttt{Int}}$	$e_1 + e_2$ is of type `Int` if e_1 and e_2 are of type `Int`.
$\dfrac{\Gamma \vdash e_1 : \texttt{t} \quad \Gamma \vdash e_2 : \texttt{t} \quad t \in \{\texttt{Int}, \texttt{Bool}\}}{\Gamma \vdash e_1 = e_2 : \texttt{Bool}}$	$e_1 = e_2$ is of type `Bool` if e_1 and e_2 are of the same type `t`, with `t` `Bool` or `Int`

Figure 1: Rules for expressions

We give the rules for some expressions in Figure 1. The exercises of this part consist in modifying the type system seen in the course to take into account the new constructs.

Answer of exercise 1

1. We provide a rule to allow for variable/type redefinition.

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash x := e \mid \Gamma[x \mapsto t]}$$

2. `x := 3`
3. `x := 3 & true`

Answer of exercise 2

1. $$\frac{\Gamma \vdash e : \texttt{Bool} \quad \Gamma \vdash S_1 \mid \Gamma_1 \quad \Gamma_1 \vdash S_2 \mid \Gamma_2 \quad \Gamma_1 = \Gamma_2}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ S\ \texttt{else}\ S\ \texttt{fi} \mid \Gamma_1}$$

2. ```
   if true then
   x := 5
   else
   x := 4
   fi
   ```
3. ```
   if x > y then
   x := 42
   else
   x := true
   fi
   ```

Answer of exercise 3

1. The rule for sequential statements says that when typing `S1; S2` with an environment Γ, one uses Γ to type statement `S1`, and it produces a new environment Γ_1. Then, environment Γ_1 is used to type statement `S2`, producing an environment Γ_2, which is the resulting environment.

$$\frac{\Gamma \vdash S_1 \mid \Gamma_1 \quad \Gamma_1 \vdash S_2 \mid \Gamma_2}{\Gamma \vdash S_1; S_2 \mid \Gamma_2}$$

2. `skip; skip`

3. The following program is incorrect because it is not possible to apply the first premise to type the first statement.

   ```
   x := 3 & true; skip
   ```

Answer of exercise 4

1. The rule for iterative statements says that when typing statement `while e do S od` in an environment Γ, one uses Γ to type `e`, which should be of type `Bool`. Moreover, `S` should be correctly-typed with environment Γ and the result of typing `S` should produce the same environment Γ. One requests that typing `S` produces the same environment because of executions "not entering the loop", i.e., when `e` evaluates to false.

$$\frac{\Gamma \vdash e : \texttt{Bool} \quad \Gamma \vdash S \mid \Gamma_1 \quad \Gamma_1 = \Gamma}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ S\ \texttt{od} \mid \Gamma}$$

2. ```
   while true do
      skip
   od
   ```
3. ```
   while 42 do
      skip
   od
   ```

Part 2: Operational Semantics ($\sim$ 7 points)

We are interested in the notion of pointer. We adapt a C-like syntactic notation. We modify the syntax of language **While** as follows:

$$\begin{array}{lcl} g & ::= & x \mid *x \mid *(x+n) \\ e & ::= & n \mid g \mid \&x \mid e \odot e \mid \texttt{Null} \\ S & ::= & g := e \mid g := \texttt{malloc}(n) \mid \texttt{free}(x) \mid \texttt{begin}\ D_V\ S\ \texttt{end} \mid S; S \\ D_V & ::= & \texttt{int}\ x\ ;\ D_V \mid \texttt{int}^*\ x\ ;\ D_V \mid \epsilon \end{array}$$

In this grammar, the constant n denotes a natural number and `Null` denotes a particular address which content is not reachable. In addition to this address `Null`, we consider two distinct addressing spaces:

- the stack: for managing local data of procedures, denoted $\texttt{Adr}_S$,
- the heap: for dynamically managing data using functions `malloc` and `free`, denoted $\texttt{Adr}_H$.

We have $\texttt{Adr}_S \cap \texttt{Adr}_H = \emptyset$.

Moreover, during the definition of the operational semantics, we consider that type verification was already done. The used types are $\{\mathsf{Int}, \mathsf{Int}*\}$.

These types have the same meaning as in language C:

- to a variable of type Int is associated an address in the stack containing a value,
- to a variable of type Int* is associated an address in the stack containing either another address in the stack or an address in the heap.

Reminder from the course

Before describing the semantics, we recall some elements seen in the course when extending language `While` with blocks and procedures.

- We consider the environment, $\texttt{Env}_V$, which associates the address of a memory location, to each variable, i.e., a symbol table:

$$\texttt{Env}_V = \texttt{Var} \to \texttt{Loc} \ni \rho, \hat{\rho}$$

- We consider the memory `Store` which associates a value to the the content of the address of a memory location:

$$\texttt{Store} = \texttt{Loc} \to \mathbb{Z} \ni \sigma$$

- For the semantics of assignments, we have seen that:

$$(x := a, \hat{\rho}, \sigma) \to \sigma[\hat{\rho}(x) \mapsto \mathcal{A}[a](\hat{\rho}, \sigma)]$$

Introducing statements `malloc` and `free` implies modifications to the semantics, from configurations to the definition of transition rules.

Moreover, we bring some modifications to the notation seen in the course: the addressing space `Loc` is renamed `Adr`.

Semantic domains

Semantic domains are modified to take into account two addressing spaces

Integers	$\mathbb{Z}$
Booleans	$\mathbb{B}$
Values	$\mathbb{Z} \cup \mathbb{B}$
Addresses	$\texttt{Adr} = \texttt{Adr}_S \cup \texttt{Adr}_H \cup \{\texttt{Null}\}$
Environments	$\texttt{Env}_V = \texttt{Var} \longrightarrow \texttt{Adr}_S$
Memory	$\texttt{Mem} = \texttt{Adr} \setminus \{\texttt{Null}\} \longrightarrow \texttt{Val} \cup \texttt{Adr}$

An address, element of `Adr`, denotes a memory location in the stack, of the heap, or `Null`. To a variable is associated an address in the stack. Moreover, we suppose known the function Γ which associates the type $\Gamma(x)$ to each variable x. This function is determined during type checking.

Configurations

Configurations for declarations and statements range in the following domains.

Declarations	$(\texttt{Dec}_V \times \texttt{Env}_V) \cup \texttt{Env}_V$
Statements	$(\texttt{Stm} \times \texttt{Env}_V^* \times \texttt{Mem}) \cup \texttt{Mem}$

Notation

- We note $\rho, \rho', \rho_S, \ldots$ the elements of $\texttt{Env}_V$, $\hat{\rho}, \hat{\rho}', \ldots$ the elements of $\texttt{Env}_V^*$. $\texttt{Env}_V^*$ denotes the set of stacks of environments as in the course.
- Finally, we note σ, σ' the elements of `Mem`. Let us note that $\sigma(\texttt{Null})$ is not defined. Said differently, the dereferencing of `Null` is forbidden.

Thus, the transitions between configurations will be of the form:

$$(S, \hat{\rho}, \sigma) \to \sigma'$$

Comments on the semantic domains As seen in the course, to an identifier x, we associate an address $\hat{\rho}(x)$ of a memory location which will contain either a value $v \in \mathbb{N}$, if x is of type Int ($\Gamma(x) = \mathsf{Int}$), or an address if x is of type Int^* ($\Gamma(x) = \mathsf{Int}^*$).

Examples Let us consider the following C program:

```
int x, *y ;
x=1;
y=&x ;
```

Here, $\hat{\rho}(x)$ and $\hat{\rho}(y)$ denote addresses in the stack, i.e, $\hat{\rho}(x), \hat{\rho}(y) \in \mathtt{Adr}_S$. We have $\sigma(\hat{\rho}(x)) = 1$ and $\sigma(\hat{\rho}(y)) = \hat{\rho}(x)$.

Let us consider the following C program:

```
int  *y ;
y = malloc(...)
*y=2;
```

We have $\sigma(\hat{\rho}(y))$ which is an address in the heap, i.e. $\sigma(\hat{\rho}(y)) \in \mathtt{Adr}_H$, and $\sigma(\sigma(\hat{\rho}(y))) = 2$.

Sub-Part 1: blocks and assignments

In the following questions, we ask to produce the semantic rules for the block construct, the declarations, and the assignment construct.

For declaration, one has to initialise variables of type `Int` to 0 and variables of type `Int`* to `Null`.

For the assignment construct, we shall decompose in several rules, according to the type of the left- and right-hand sides in the assignment.

Answer of exercise 5

1. $$\frac{(D_V, \emptyset, \sigma) \rightarrow_D (\rho_l, \sigma') \quad (S, \hat{\rho} \oplus \rho_l, \sigma') \rightarrow \sigma''}{(\mathsf{begin}\ D_V\ S\ \mathsf{end}, \hat{\rho}, \sigma) \rightarrow \sigma''}$$

2. $$\frac{(D_V, \rho_l[x \mapsto a_x], \sigma[a_x \mapsto 0]) \rightarrow (\rho_l', \sigma')}{(\mathtt{int}\ x\ ; D_V, \rho_l, \sigma) \rightarrow (\rho_l', \sigma')} \quad \frac{(D_V, \rho_l[x \mapsto a_x], \sigma[a_x \mapsto \mathtt{Null}]) \rightarrow (\rho_l', \sigma')}{(\mathtt{int}^*\ x\ ; D_V, \rho_l, \sigma) \rightarrow (\rho_l', \sigma')}$$

3. $$\frac{}{(x := y, \hat{\rho}, \sigma) \rightarrow \sigma[\hat{\rho}(x) \mapsto \sigma(\hat{\rho}(y))]}$$

4. $$\frac{}{(x := y, \hat{\rho}, \sigma) \rightarrow \sigma[\hat{\rho}(x) \mapsto \sigma(\hat{\rho}(y))]}$$

5. $$\frac{}{(x := *y, \hat{\rho}, \sigma) \rightarrow \sigma[\hat{\rho}(x) \mapsto \sigma \circ \sigma \circ \hat{\rho}(y)]}$$

6. $$\frac{}{(x = \&y, \hat{\rho}, \alpha, \sigma) \rightarrow \sigma[\hat{\rho}(x) \mapsto \hat{\rho}(y)]}$$

Sub-Part 2 : `malloc` and `free`

We are interested in primitives `malloc` and `free`. For this, we modify the memory domain by associating, to an address, either an address in the stack or an address in the heap and a size.

We replace $\texttt{Mem} = \texttt{Adr} \setminus \{\texttt{Null}\} \longrightarrow \texttt{Val} \cup \texttt{Adr}$ by

$$\texttt{Mem} = \texttt{Adr} \setminus \{\texttt{Null}\} \longrightarrow \texttt{Val} \cup \texttt{Adr}_S \cup (\texttt{Adr}_H \times \mathbb{N}) \cup \{\texttt{Null}\}$$

We suppose having a system allocator `new` which, called with a natural number as parameter, returns an address of a free location in the heap.

Answer of exercise 6

1.
$$\frac{v = \mathcal{N}(n) \quad l_h = \mathsf{new}(v)}{(x = \mathsf{malloc}(n), \hat{\rho}, \sigma) \rightarrow (\sigma[\hat{\rho}(x) \mapsto (l_h, v)])}$$

Answer of exercise 7

1.
$$\frac{\sigma(\hat{\rho}(x)) \in \texttt{Adr}_H \times \mathbb{N}}{(\mathsf{free}(x), \hat{\rho}, \sigma) \rightarrow \sigma}$$

2.
$$\frac{\sigma(\hat{\rho}(x)) \in \texttt{Adr}_H \times \mathbb{N}}{(\mathsf{free}(x), \hat{\rho}, \sigma) \rightarrow \sigma[\hat{\rho}(x) \mapsto \texttt{Null}]}$$

Answer of exercise 8

1. (partial)
$$\frac{\sigma(\hat{\rho}(p)) = (l_p, v_p) \quad \sigma(\hat{\rho}(q)) = (l_q, v_q)}{(*p = *q, \hat{\rho}, \sigma) \rightarrow \sigma[l_p \mapsto \sigma(l_q)]}$$

2. (partial)
$$\frac{\sigma(\hat{\rho}(p)) = (l_p, v_p) \quad \sigma(\hat{\rho}(q)) = (l_q, v_q) \quad v_n = \mathcal{N}(n) \quad v_m = \mathcal{N}(m) \quad v_n < v_p \quad v_m < v_q}{(*(p+n) = *(q+m), \hat{\rho}, \sigma) \rightarrow \sigma[(l_p + v_n) \mapsto \sigma(l_q + v_m)]}$$

Part 3 : Optimisation ($\sim$ 5 points)

We consider the control flow graph in Figure **??**. We are interested in available expressions and live variables, in the sense given in the course.

In the following, we consider the modified graph resulting from the previous exercise.

Answer of exercise 9

1. The following table presents the solution to the two first questions.
2.

				*Step*1		*Step*2		*Step*3		*Step*4	
B	pre(B)	gen(B)	kill(B)	in(B)	out(B)	in(B)	out(B)	in(B)	out(B)	in(B)	out(B)
B_1	$\emptyset$	$\emptyset$	Σ	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
B_2	B_1, B_4	$\emptyset$	$\emptyset$	Σ	Σ	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
B_3	B_2	Σ	$\emptyset$	Σ	Σ	Σ	Σ	$\emptyset$	Σ	$\emptyset$	Σ
B_4	B_3	$\emptyset$	Σ	Σ	$\emptyset$	Σ	$\emptyset$	Σ	$\emptyset$	Σ	$\emptyset$
B_5	B_2	e_2	Σ	Σ	e_2	Σ	e_2	$\emptyset$	e_2	$\emptyset$	e_2

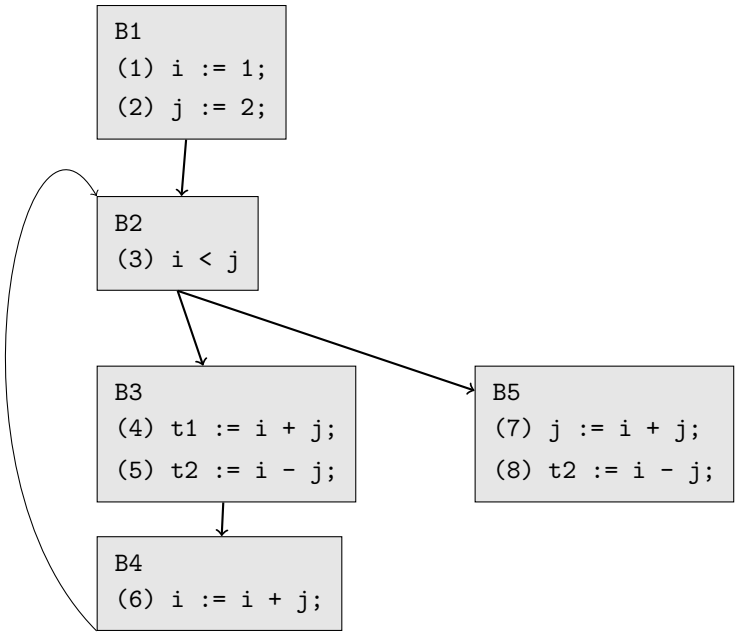
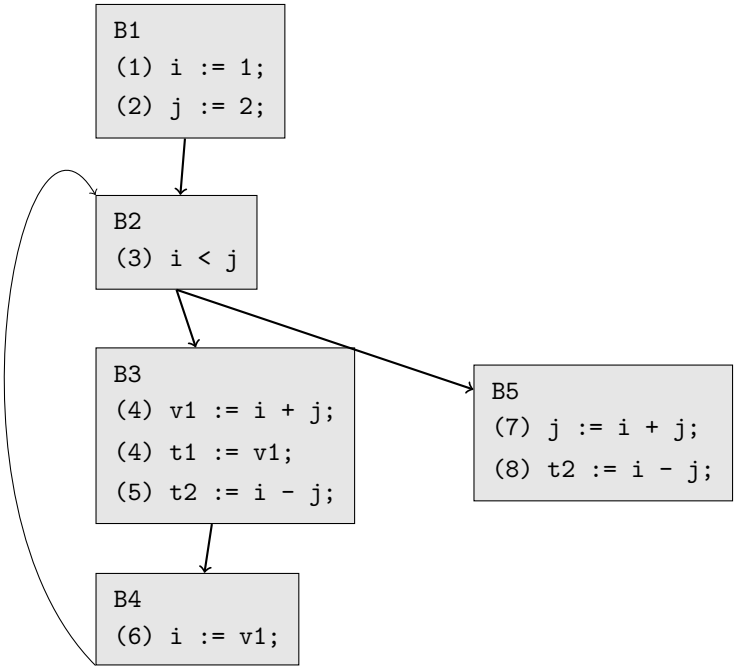


Figure 2: Initial control flow graph

3. Expression $i + j$ is available at the entrance of B4. Moreover, this expression is computed in this block and every assigment to its variables happens after the first computation. We thus introduce a new temporary variable $v1$ to store the value of $i + j$ during its first computation and modify the CFG accordingly, as seen in the course. The modified CFG is depicted below.



Answer of exercise 10

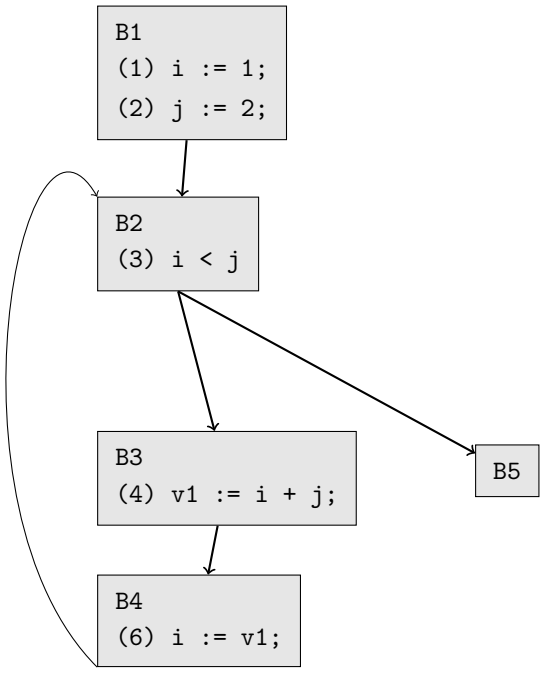
1. We consider the set of variables $\Sigma = \{i, j, t1, t2, v1\}$. The following table presents the solution to the

two first questions.

2.

				*Step*1		*Step*2		*Step*3		*Step*4	
B	succ(B)	gen(B)	kill(B)	out(B)	in(B)	out(B)	in(B)	out(B)	in(B)	out(B)	in(B)
B_1	B_2	$\emptyset$	i, j	$\emptyset$	$\emptyset$	i, j	$\emptyset$	i, j	$\emptyset$	i, j	$\emptyset$
B_2	B_3, B_5	i, j	$\emptyset$	$\emptyset$	i, j	i, j	i, j	i, j	i, j	i, j	i, j
B_3	B_4	i, j	v_1, t_1, t_2	$\emptyset$	i, j	v_1	i, j	v_1, j	i, j	v_1, j	i, j
B_4	B_2	v_1	i	$\emptyset$	v_1	i, j	v_1, j	i, j	v_1, j	i, j	v_1, j
B_5	$\emptyset$	i, j	t_2, j	$\emptyset$	i, j	$\emptyset$	i, j	$\emptyset$	i, j	$\emptyset$	i, j

3. The modified CFG is depicted below.



Part 4 : Code Generation ($\sim$ 5 points)

Answer of exercise 11

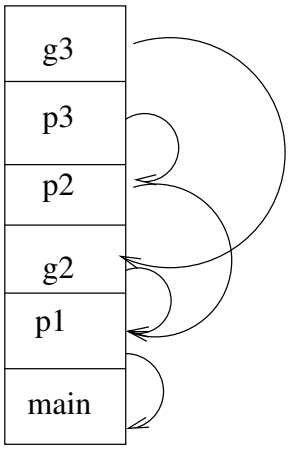
1. Call tree: `main -> p1 -> g2 -> p2 -> p3 -> g3 (as p)`. The call stack is as usual, in a procedure p the dynamic link points to the definition environment of the caller of p and the static link points to the definition environment of the procedure where p is defined (represented below).

```
#include <stdio.h>
typedef int (*intprocint) ( int);
/* intprocint denotes the type procedure with an integer parameter and an integer result */

typedef int (*intprocintint) (int,int);
typedef int (*intprocvoid) (void);

main(){
  int x1;
  int p1 () {
    int x ;
    int y ;
    int z ;
    int p2(int x,intprocint p) {
      int p3 (intprocint p){
        return p(x+x1+y);
      }
      z = 11;
      z = y + p3(p);
      return z;
    }
    int g2(int x) {
      int g3 (int x){return (x-1);}
      if (x>0) y=p2(x1,g3);
    }
    y=22;
    z=g2(2);
    return z;
  }
  x1=0;
  x1=p1();
  printf("%d\n",x1);
}
```

Figure 3: Program for code generation



2.
```
!z = g2(2)
ADD R1, R0, 2
push (R1) ! push the parameter 2
ADD SP, SP, -4 ! reserve space for the return value
push(FP) ! SL of g2
call main.p1.g2 ! calling main.p1.g2
ADD SP, SP, +4 ! pop SL of call main.p1.g2
LD R2 [SP], ! retrieving return value
ST R2, [FP - 12] !store return value to z (with offset -12)
ADD SP, SP, +8 ! pop return value and parameter
```
3.
```
! z = y + p3(p)
LD R1, [FP + 8] ! 1 indirection on the SL to get the DL of p1 for y
LD R1, [R1 - 8] ! load y with offset -8
LD R2, [FP + 20] ! retrieve the address of p as parameter
push(R2) ! pushing address of p
LD R3, [FP + 16] ! retrieve SL of p
push(R3) ! push SL of p
ADD SP, SP, -4 ! reserve space for the return value
push(FP) ! push the SL of main.p1.p2.p3
call main.p1.p2.p3 ! calling main.p1.p2.p3
ADD SP, SP, +4 ! pop SL of main.p1.p2.p3
LD R2, [SP] ! retrieving return value
ADD R1, R1, R2 ! y + return value of main.p1.p2.p3
ADD SP, SP, 12 ! pop return value
LD R4, [FP + 8] ! 1 indirection on the SL to get the DL of p1 where z is defined
ST R1, [R4 - 12] ! store in z, with offset -12
```
4.
```
! return p (x + x1 + y)
LD R1, [FP + 8] ! 2 indirections on the SL to get the DL of p1 where x is defined
LD R1, [R1 + 8]
LD R2, [R1 - 4] ! loading the value of x with offset -4
LD R3, [R1 + 8] ! 1 additional indirection to get the DL of main
                  (R1 already contains the DL of main.p1)
LD R3, [R3 - 4] ! load x1
LD R4, [R1 - 8] ! load y with offset -8
ADD R2, R2, R3 ! x + x1
ADD R2, R2, R4 ! x + x1 + y
push(R2) ! pushing parameter
ADD SP, SP, -4 ! reserve space for the return value
LD R5, [FP + 16] ! load the SL of p from parameters
push(R5) ! push SL of p
LD R6, [FP + 20] ! load address of p from parameters
call R6
ADD SP, SP, +4 ! pop SL of p
LD R2, [SP]
ST R2, [FP + 12] ! store return value of p as return value of p3
ADD SP, SP, +8 ! pop return value of p and parameter
```
5.
```
! y = p2 (x1, g3)
LD R1, [FP + 8] ! 2 indirections on the SL to get the DL of main
LD R1, [R1 + 8]
LD R2, [R1 - 4] ! load x1 with offset -4
push(R2) ! pushing first parameter
```

```
SET R3 main.p1.g2.g3 ! set address of main.p1.g2.g3 in a register
push(R3) ! pushing the address of main.p1.g2.g3 as a parameter
push(FP) ! pushing the SL of main.p1.g2.g3
ADD SP, SP, -4 ! reserve space for the result
push(R1) ! pushing SL of main.p1.p2, which is the DL of p1
call main.p1.p2
ADD SP, SP, +4 ! pop SL of p2
LD R2, [SP]
ST R2, [R1 - 8] ! storing return value to y, with offset -8 in the environment of p1
ADD SP, SP, +16 ! pop return value of p2 and parameters
```