La transpilation de Fortran vers C

Présentation de **Erwan FALAUX-BACHELOT**

travail réalisé avec Yann MIQUEL–ERDMANN

Introduction

Le Fortran : F
Créé par : IMB en 1954
Utilisé surtout dans les supercalculateurs

Problématique

Comment implémenter une conversion rapide de programmes Fortran en programmes C ?

Les expressions régulières

- Expressions régulières
- Automates

définies inductivement sur :

$$\emptyset,\ \varepsilon,\ a \in \Sigma$$

Analyse Lexicale → Analyse Syntaxique → Syntaxe Abstraite → Conversion

Les expressions régulières

- Expressions régulières
- Automates

définies inductivement sur :

$$\emptyset,\ \varepsilon,\ a \in \Sigma$$

avec les règles usuelles :

$$\cdot,\ |,\ *$$

Analyse Lexicale → Analyse Syntaxique → Syntaxe Abstraite → Conversion

Les expressions régulières

- Expressions régulières
- Automates

définies inductivement sur :

$\emptyset,\ \varepsilon,\ a \in \Sigma$

avec les règles usuelles :

$\cdot,\ |,\ *$

et des additionnelles :

$+,\ ?,\ [a-z],\ \sim$

Analyse Lexicale → Analyse Syntaxique → Syntaxe Abstraite → Conversion

Les expressions régulières

- Expressions régulières
- Automates

définies inductivement sur :

$\emptyset,\ \varepsilon,\ a \in \Sigma$

avec les règles usuelles :

$\cdot,\ |,\ *$

et des additionnelles :

$+,\ ?,\ [a - z],\ \sim$

exemple :

[0 - 9]+ (.[0 - 9]+) ?

Analyse Lexicale → Analyse Syntaxique → Syntaxe Abstraite → Conversion

Les expressions régulières

- Expressions régulières
- Automates

exemple :

$\boxed{[0\text{ - }9]+}\,(.[0\text{ - }9]+)\,?$

définies inductivement sur :

$\emptyset,\ \varepsilon,\ a \in \Sigma$

avec les règles usuelles :

$\cdot,\ |,\ *$

et des additionnelles :

$+,\ ?,\ [a-z],\ \sim$

Analyse Lexicale → Analyse Syntaxique → Syntaxe Abstraite → Conversion

Les expressions régulières

- Expressions régulières
- Automates

définies inductivement sur :

$\emptyset,\ \varepsilon,\ a \in \Sigma$

avec les règles usuelles :

$\cdot,\ |,\ *$

et des additionnelles :

$+,\ ?,\ [a-z],\ \sim$

exemple :

$\boxed{[0 - 9]+}\boxed{(.[0 - 9]+)\,?}$

Analyse Lexicale → Analyse Syntaxique → Syntaxe Abstraite → Conversion

Les automates

- Expressions régulières
- Automates

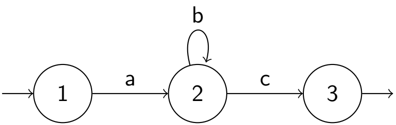
$$(\Sigma, Q, I, F, \delta)$$

Analyse Lexicale → Analyse Syntaxique → Syntaxe Abstraite → Conversion

Les automates

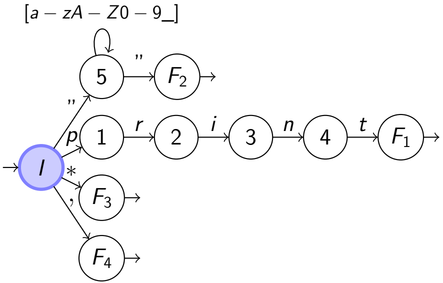
- Expressions régulières
- Automates

$$(\Sigma, Q, I, F, \delta)$$



automate pour $a \cdot b * \cdot c$

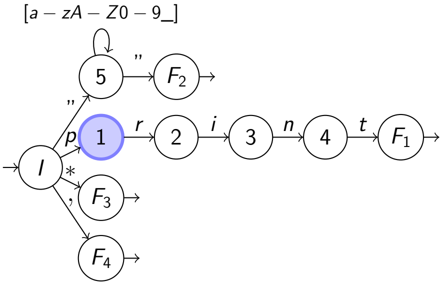
Les automates : étude d'un exemple




```
print*,"Hello World"
```

```
res = [ ]
```

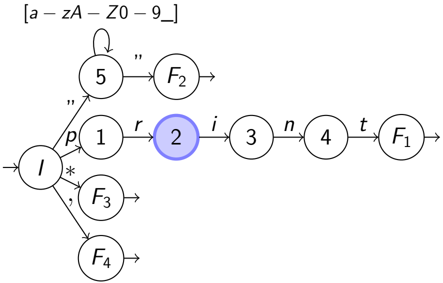
Les automates : étude d'un exemple




```
print*,"Hello World"
```

```
res = [ ]
```

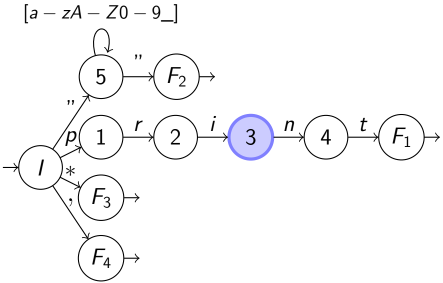
Les automates : étude d'un exemple



`print*,"Hello World"`

`res = [ ]`

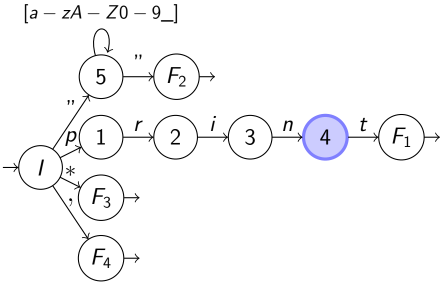
Les automates : étude d'un exemple



`print*,"Hello World"`

`res = [ ]`

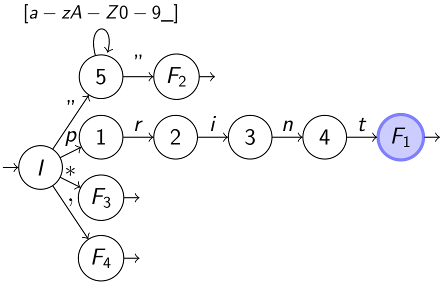
Les automates : étude d'un exemple




```
print*,"Hello World"
```

```
res = [ ]
```

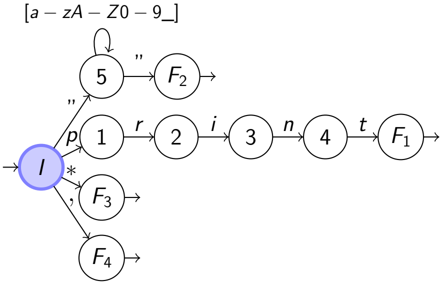
Les automates : étude d'un exemple




```
print*,"Hello World"
```

```
res = [ ]
```

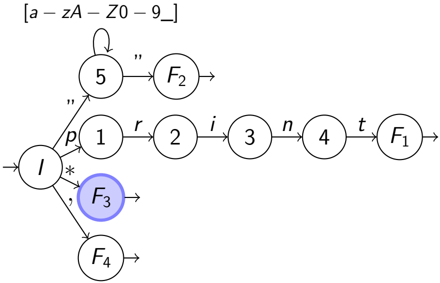
Les automates : étude d'un exemple



`print*,"Hello World"`

`res = [ 'print' ]`

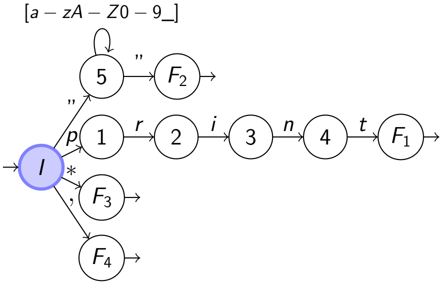
Les automates : étude d'un exemple




```
print*,"Hello World"
```

```
res = [ 'print' ]
```

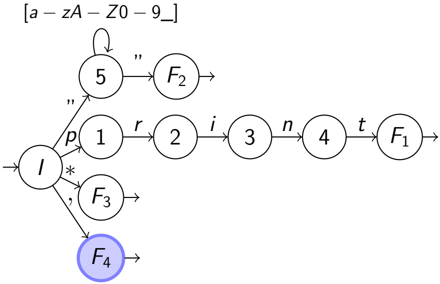
Les automates : étude d'un exemple




```
print*,"Hello World"
```

```
res = [ 'print' , '*' ]
```

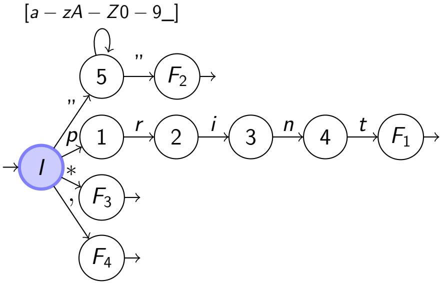
Les automates : étude d'un exemple



`print*,"Hello World"`

`res = [ 'print' , '*' ]`

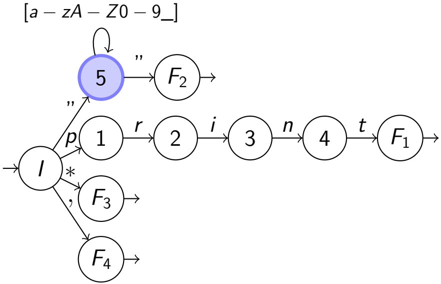
Les automates : étude d'un exemple



`print*,"Hello World"`

`res = [ 'print' , '*' , ',' ]`

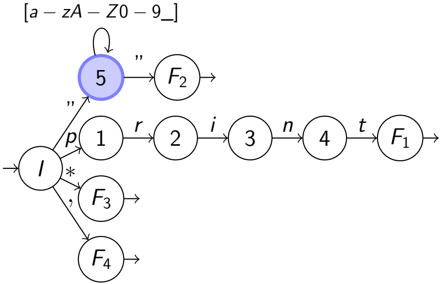
Les automates : étude d'un exemple




```
print*,"Hello World"
```

```
res = [ 'print' , '*' , ',' ]
```

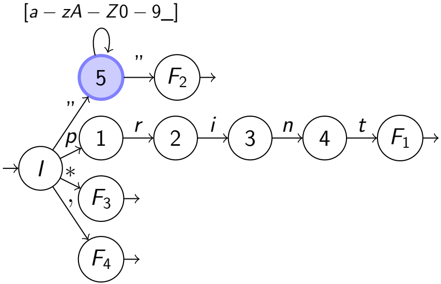
Les automates : étude d'un exemple




```
print*,"Hello World"
```

```
res = [ 'print' , '*' , ',' ]
```

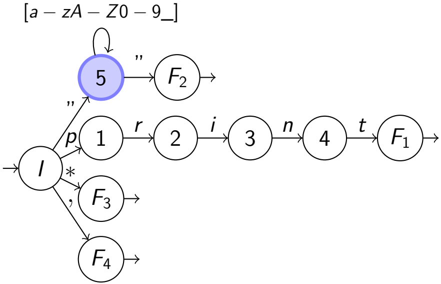
Les automates : étude d'un exemple




```
print*,"Hello World"
```

```
res = [ 'print' , '*' , ',' ]
```

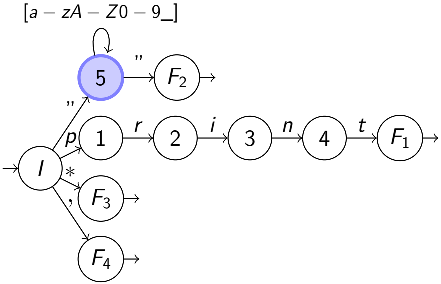
Les automates : étude d'un exemple



print*,"Hello World"

res = ['print' , '*' , ',']

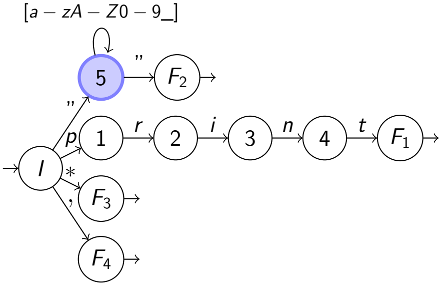
Les automates : étude d'un exemple




```
print*,"Hello World"
```

```
res = [ 'print' , '*' , ',' ]
```

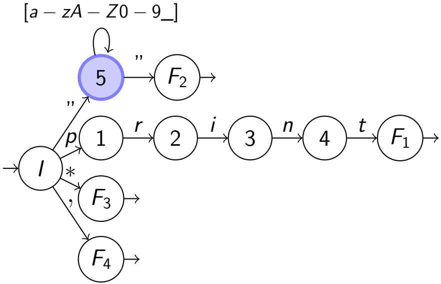
Les automates : étude d'un exemple



print*,"Hello World"

res = ['print' , '*' , ',']

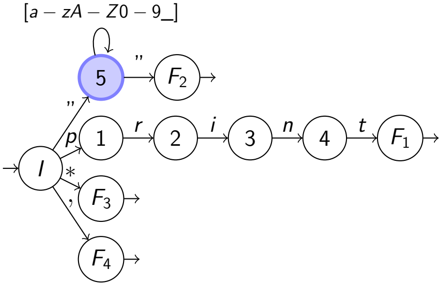
Les automates : étude d'un exemple



`print*,"Hello World"`

`res = [ 'print' , '*' , ',' ]`

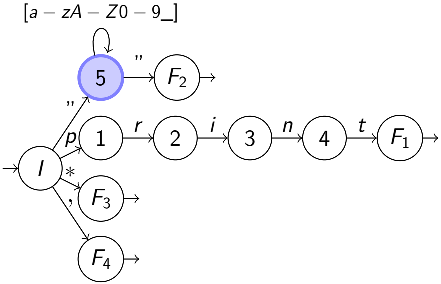
Les automates : étude d'un exemple




```
print*,"Hello World"
```

```
res = [ 'print' , '*' , ',' ]
```

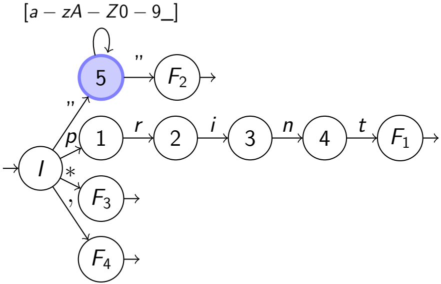
Les automates : étude d'un exemple



print*,"Hello World"

res = ['print' , '*' , ',']

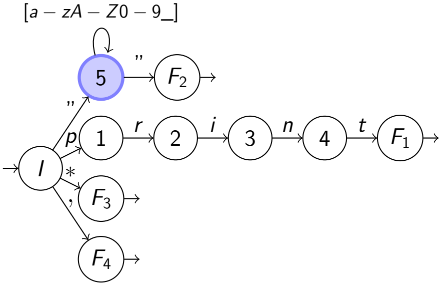
Les automates : étude d'un exemple




```
print*,"Hello World"
```

```
res = [ 'print' , '*' , ',' ]
```

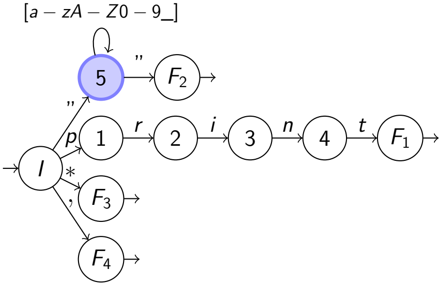
Les automates : étude d'un exemple



`print*,"Hello World"`

`res = [ 'print' , '*' , ',' ]`

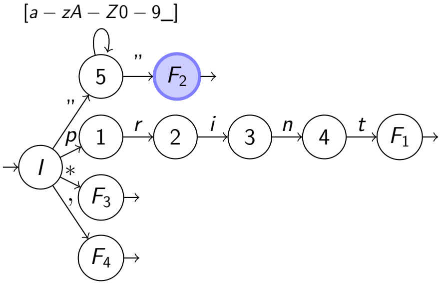
Les automates : étude d'un exemple



print*,"Hello World"

res = ['print' , '*' , ',']

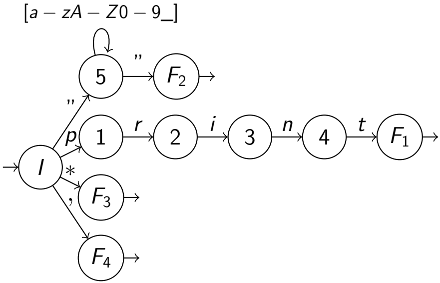
Les automates : étude d'un exemple



`print*,"Hello World"`

`res = [ 'print' , '*' , ',' ]`

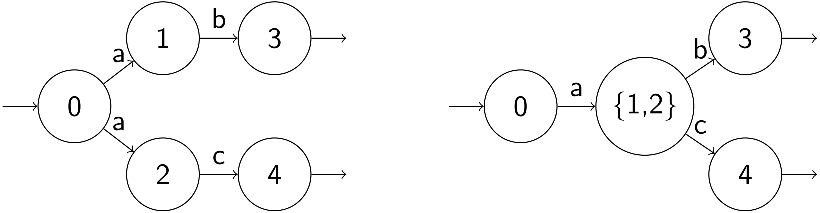
Les automates : étude d'un exemple

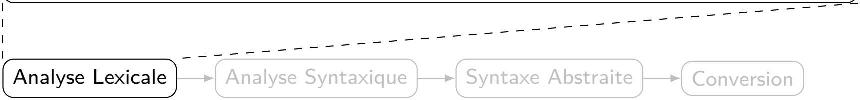


`print*,"Hello World"`

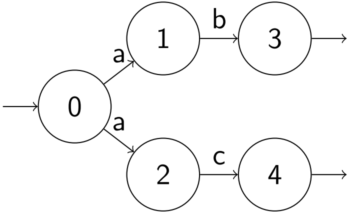
`res = [ 'print' , '*' , ',' , '"Hello World"' ]`

La déterminisation





La déterminisation

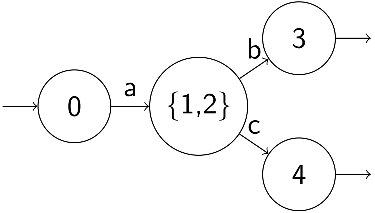



```
type automate = {
  nodes : int list;
  debut_l : int list;
  fin : (int * terminal) list;
  transitions : (char option * int) list array;
}
```

Analyse Lexicale → Analyse Syntaxique → Syntaxe Abstraite → Conversion

La déterminisation

```
type automate_det = {
  nodes : int list;
  debut : int;
  fin : terminal option
    ↪  array;
  transitions : int array
    ↪  array;
}
```



Analyse Syntaxique

- Grammaire
- LL1

Règles de production :

`A -> B`

`A ->` $c, c \in \Sigma^*$

Analyse Lexicale → Analyse Syntaxique → Syntaxe Abstraite → Conversion

Analyse Syntaxique

- Grammaire
- LL1

Règles de production :

A -> B

A -> $c, c \in \Sigma^*$

Exemple :

```
S -> A
A -> aA
A -> B
B -> bB
B -> epsilon
```

Analyse Lexicale → **Analyse Syntaxique** → Syntaxe Abstraite → Conversion

Analyse Syntaxique

- Grammaire
- LL1

```
      S
      |
      A
     / \
    a   A
        |
        B
       / \
      b   B
          |
          ε
```

Arbre de dérivation de ab

Règles de production :

`A -> B`

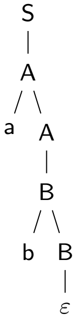
`A ->` $c, c \in \Sigma^*$

Exemple :

```
S -> A
A -> aA
A -> B
B -> bB
B -> epsilon
```

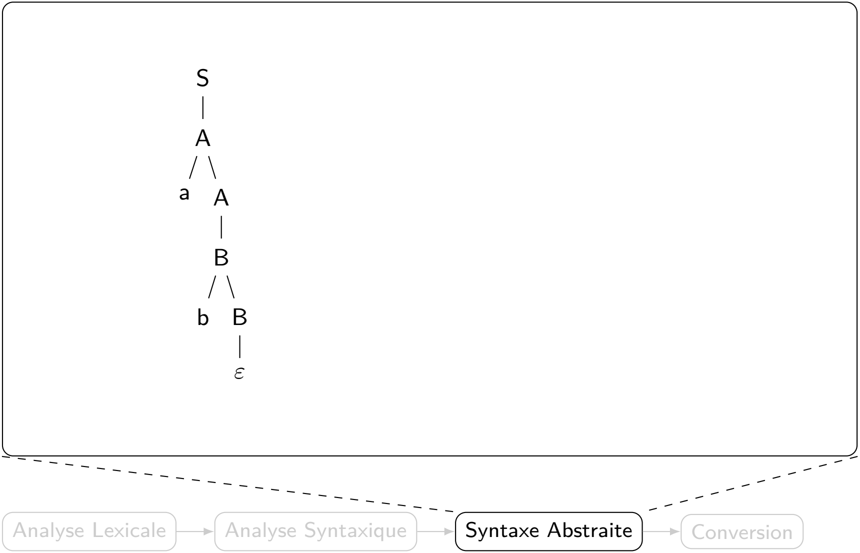
- Grammaire
- LL1

```
                 S -> A
                 A -> aA
[a, b]  +        A -> B          =>
                 B -> bB
                 B -> epsilon
```

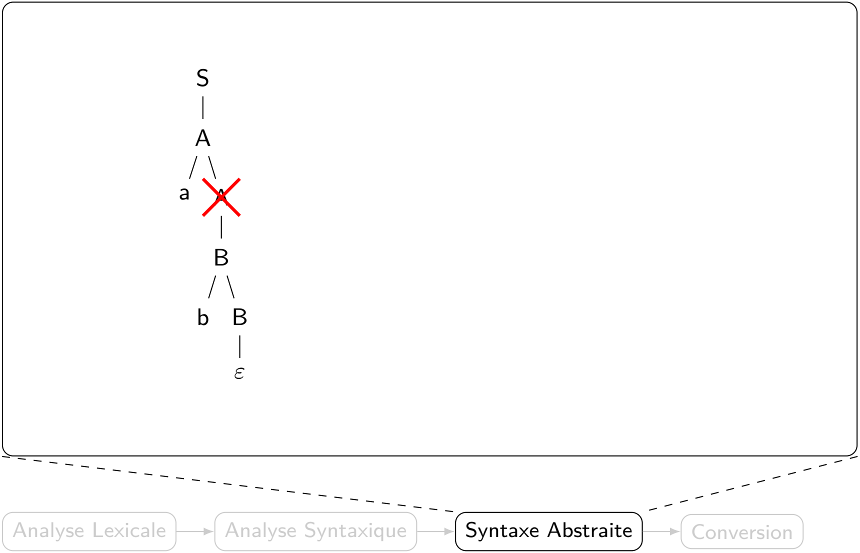


Analyse Lexicale → **Analyse Syntaxique** → Syntaxe Abstraite → Conversion

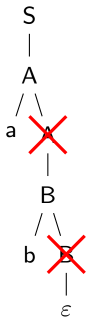
Syntaxe abstraite



Syntaxe abstraite

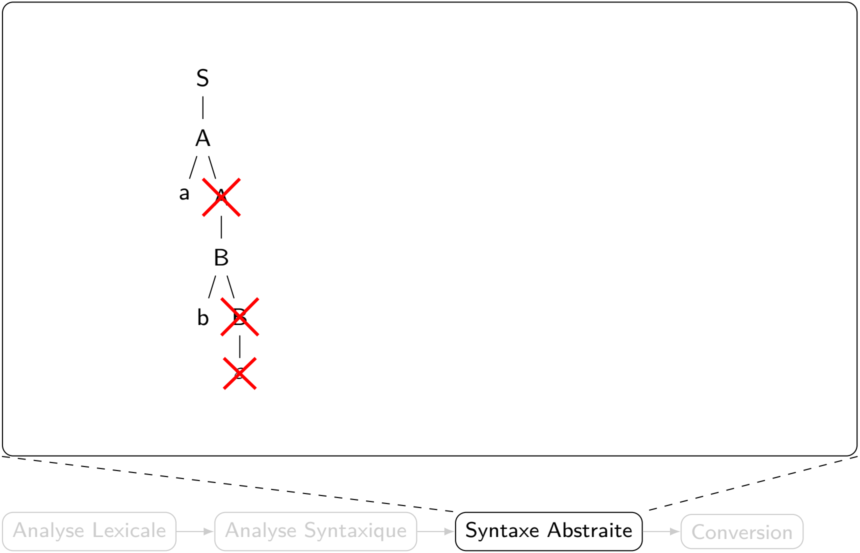


Syntaxe abstraite

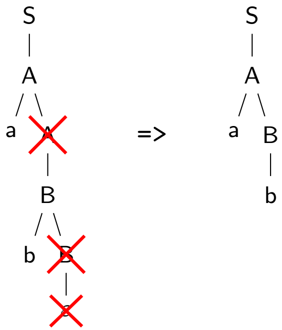


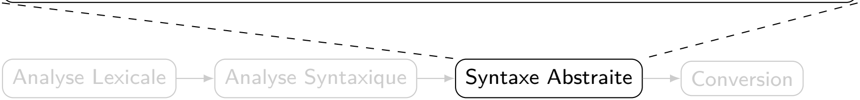
Analyse Lexicale → Analyse Syntaxique → **Syntaxe Abstraite** → Conversion

Syntaxe abstraite

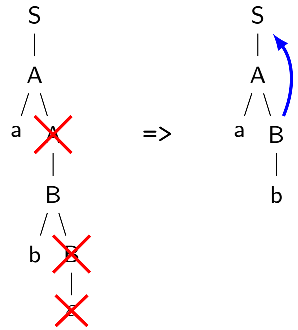


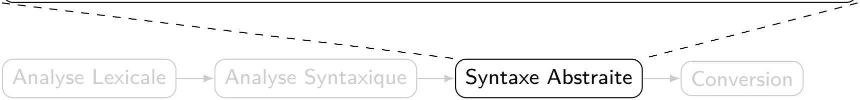
Syntaxe abstraite



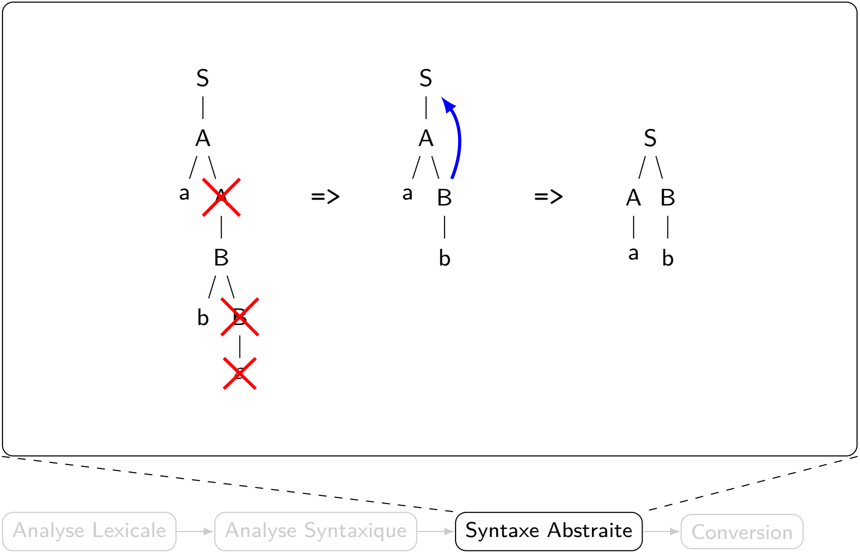


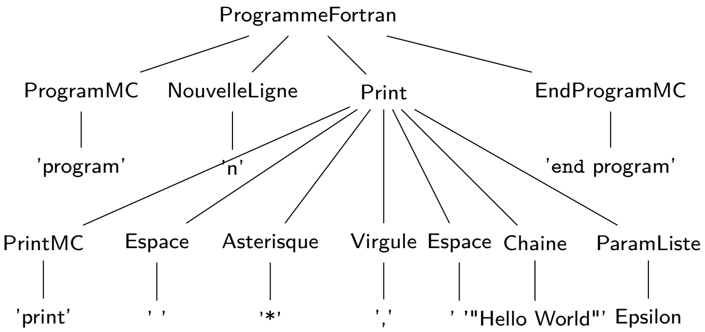
Syntaxe abstraite

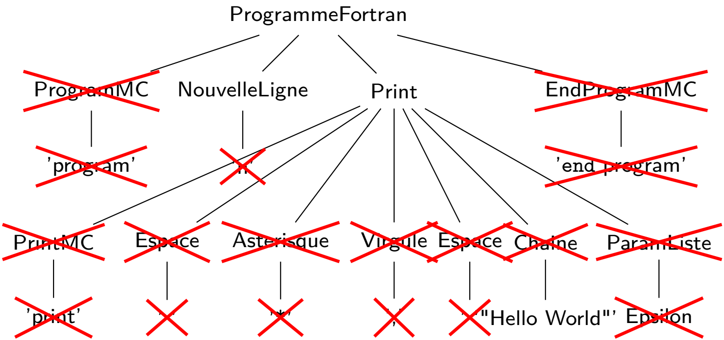


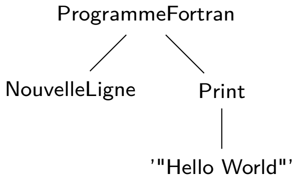


Syntaxe abstraite




ProgrammeFortran
ProgramMC
NouvelleLigne
Print
EndProgramMC
'program'
'n'
'end program'
PrintMC
Espace
Asterisque
Virgule
Espace
Chaine
ParamListe
'print'
' '
'*'
','
' '
'"Hello World"'
Epsilon


ProgrammeFortran
ProgramMC
NouvelleLigne
Print
EndProgramMC
'program'
'end program'
PrintMC
Espace
Asterisque
Virgule
Espace
Chaîne
ParamListe
'print'
'"Hello World"'
Epsilon


ProgrammeFortran
NouvelleLigne
Print
'"Hello World"'

1. Analyse Lexicale

2. Analyse Syntaxique

3. Convertion vers la syntaxe abstraite

4. Traduction vers le langage de sortie

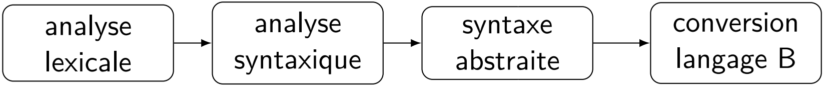
Traduction

parcours en profondeur

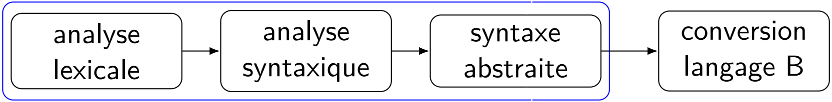
convertion en chaîne

Analyse Lexicale → Analyse Syntaxique → Syntaxe Abstraite → **Conversion**

Transpileur

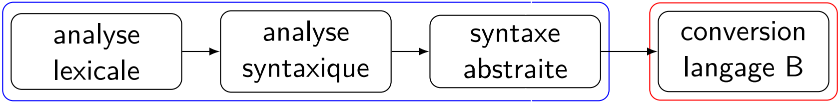


Transpileur



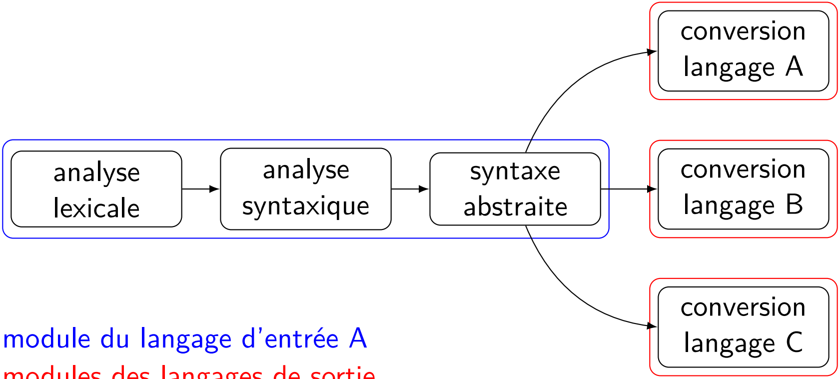
module du langage d'entrée A

Transpileur



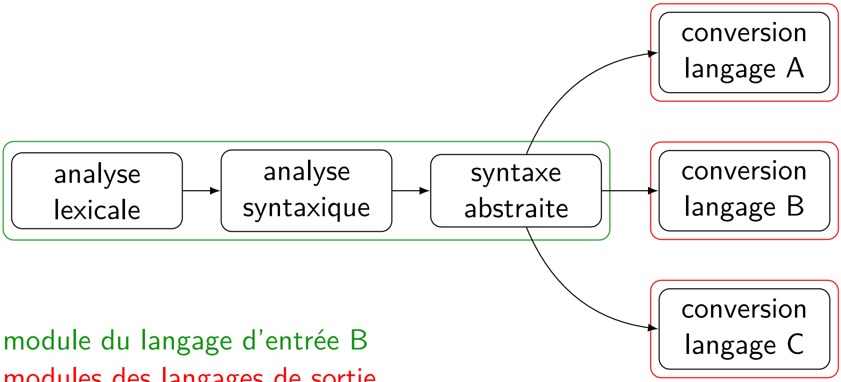
module du langage d'entrée A
modules du langage de sortie

Transpileur



module du langage d'entrée A
modules des langages de sortie

Transpileur



module du langage d'entrée B
modules des langages de sortie

Conclusion

- Conversion rapide
- Partie automatisée rend la création moins pénible
- Processus interchangeable avec les langages souhaités

```
type operateur = Plus | Moins | Fois | Division |
↪   Puissance | Assignation

type comparateur =
    | PlusPetit
    | PlusGrand
    | StrictPlusPetit
    | StrictPlusGrand
    | Egal
    | NonEgal

type operateur_logique = Et | Ou | Non | Equivalent
↪    | NonEquivalent

type syntax =
    | Any
    | Character
```

```
    | Complex
    | Constant
    | Do
    | Double_precision
    | Else
    | Else_if
    | Function
    | If
    | End_if
    | Integer
    | Logical
    | Program
    | Real
    | While
    | Call
    | Print
    | Return
```

```
  | Size
  | For
  | Step
  (* v---only for post-processing---v *)
  | Out
  | Subroutine

type token =
  | Operateur of operateur
  | Comparateur of comparateur
  | OperateurLogique of operateur_logique
  | Syntax of syntax
  | NewLine
  | Name of string
  | Integer of string
  | Floating of string
  | Double of string
```

```
    | Chaine of string
    | Commentaire of string
    | Booleen of bool
    | Parentheseouvrante
    | Parenthesefermante
    | ProgramRoot
    | ToFlatten

(* paramètres et enfants confondus dans la liste
↪  *)
type ast = Noeud of token * ast list

let string_of_token (t : token) : string =
  match t with
    | Operateur Plus -> "Operateur Plus"
    | Operateur Moins -> "Operateur Moins"
    | Operateur Fois -> "Operateur Fois"
```

```
    | Operateur Division -> "Operateur Division"
    | Operateur Puissance -> "Operateur Puissance"
    | Operateur Assignation -> "Operateur Assignation"
    | Comparateur PlusPetit -> "Comparateur PlusPetit"
    | Comparateur PlusGrand -> "Comparateur PlusGrand"
    | Comparateur StrictPlusPetit -> "Comparateur StrictPlusPetit"
    | Comparateur StrictPlusGrand -> "Comparateur StrictPlusGrand"
    | Comparateur Egal -> "Comparateur Egal"
    | Comparateur NonEgal -> "Comparateur NonEgal"
    | OperateurLogique Et -> "OperateurLogique Et"
    | OperateurLogique Ou -> "OperateurLogique Ou"
    | OperateurLogique Non -> "OperateurLogique Non"
```

```
    | OperateurLogique Equivalent -> "OperateurLogique Equivalent"
    | OperateurLogique NonEquivalent -> "OperateurLogique NonEquivalent"
    | Syntax Any -> "Syntax Any"
    | Syntax Character -> "Syntax Character"
    | Syntax Complex -> "Syntax Complex"
    | Syntax Constant -> "Syntax Constant"
    | Syntax Do -> "Syntax Do"
    | Syntax Double_precision -> "Syntax Double_precision"
    | Syntax Else -> "Syntax Else"
    | Syntax Else_if -> "Syntax Else_if"
    | Syntax Function -> "Syntax Function"
    | Syntax If -> "Syntax If"
    | Syntax End_if -> "Syntax End_if"
    | Syntax Out -> "Syntax Out"
```

```
    | Syntax Integer -> "Syntax Integer"
    | Syntax Logical -> "Syntax Logical"
    | Syntax Program -> "Syntax Program"
    | Syntax Real -> "Syntax Real"
    | Syntax Subroutine -> "Syntax Subroutine"
    | Syntax While -> "Syntax While"
    | Syntax Call -> "Syntax Call"
    | Syntax Print -> "Syntax Print"
    | Syntax Return -> "Syntax Return"
    | Syntax Size -> "Syntax Size"
    | Syntax For -> "Syntax For"
    | Syntax Step -> "Syntax Step"
    | NewLine -> "NewLine"
    | Name s -> "Name " ^ s
    | Integer s -> "Integer " ^ s
    | Floating s -> "Floating " ^ s
    | Double s -> "Double " ^ s
```

```
    | Chaine s -> "Chaine " ^ s
    | Commentaire s -> "Commentaire " ^ s
    | Booleen b -> "Booleen " ^ string_of_bool b
    | Parentheseouvrante -> "Parentheseouvrante"
    | Parenthesefermante -> "Parenthesefermante"
    | ProgramRoot -> "ProgramRoot"
    | ToFlatten -> "ToFlatten"

let print_token (t : token) : unit = print_endline
 ↪  (string_of_token t)
```

```ocaml
type libs = string list

let add_lib (l : libs) (name : string) : libs =
  if not (List.mem name l) then name :: l else l
```

```
open Abstract_tokens

type environnement = (string, token) Hashtbl.t
type intent = In | Out | InOut
type var_type_subroutine = (string, intent)
↪  Hashtbl.t

let print_env (env: environnement): unit =
  Hashtbl.iter (fun k v -> print_string k ;
   ↪  print_string " -> " ; print_token v;
   ↪  print_newline ()) env


let rec last_of_list (l: 'a list): 'a =
  match l with
   | [] -> failwith "Liste vide "
   | e::[] -> e
```

```
  | e::q -> last_of_list q


(** crée un environnement à partir de l'ast [t] *)
let create_env_from_ast (t : ast) : environnement =
  let env = Hashtbl.create 0 in
  (** ajoute récursivement sur [t] les variables dans l'environnement *)
  let rec add_env (t : ast) : unit =
    match t with
    | Noeud (Syntax Double_precision, [ Noeud (Name s, []) ])
    | Noeud
        ( Syntax Double_precision,
          [ Noeud (Operateur Assignation, [ Noeud (Name s, []); _ ]) ] ) ->
        Hashtbl.add env s (Syntax Double_precision)
```

```
    | Noeud (Syntax Integer, [ Noeud (Name s, []) ])
    | Noeud
        ( Syntax Integer,
          [ Noeud (Operateur Assignation, [ Noeud (Name s, []); _ ]) ] ) ->
        Hashtbl.add env s (Syntax Integer)
    | Noeud (Syntax Real, [ Noeud (Name s, []) ])
    | Noeud
        ( Syntax Real,
          [ Noeud (Operateur Assignation, [ Noeud (Name s, []); _ ]) ] ) ->
        Hashtbl.add env s (Syntax Real)
    | Noeud (Syntax Logical, [ Noeud (Name s, []) ])
    | Noeud
        ( Syntax Logical,
```

```
          [ Noeud (Operateur Assignation, [ Noeud (Name s, []); _ ]) ] ) ->
        Hashtbl.add env s (Syntax Real)
    | Noeud (Syntax Character, [ Noeud (Syntax Size, _); Noeud (Name s, []) ])
    | Noeud (Syntax Character, [ Noeud (Name s, []) ])
    | Noeud
        ( Syntax Character,
          [
            Noeud
              ( Operateur Assignation,
                [ Noeud (Syntax Size, _); Noeud (Name s, []); _ ] );
          ] )
    | Noeud
        ( Syntax Character,
```

```
            [ Noeud (Operateur Assignation, [ Noeud (Name s, []); _ ]) ] ) ->
          Hashtbl.add env s (Syntax Real)
      | Noeud (Syntax Function, Noeud(Name n,[])::l) ->(
          List.iter add_env l;
          match last_of_list l with
          | Noeud(Syntax Return , Noeud(Name v,[])::_) ->  Hashtbl.add env n (Hashtbl.find env v)
          | _ -> () (* pas de return à la fin de la fct donc c'est une subroutine *)
        )

      | Noeud (_, l) -> List.iter add_env l
  in
  add_env t;
```

```
  env

(** déduis de l'utilisation des variables dans [t] si elles sont in, out out *)
let create_subroutine_intent (t : ast) : var_type_subroutine =
  let sub_intent = Hashtbl.create 0 in
  (** traite récursivement les variables sur [t] *)
  let rec subroutine_intent_aux (t : ast) : unit =
    (* TODO ajouter lorsque la syntaxe abstraite pour le in/out/inout est ok *)
    match t with
    | Noeud (Operateur Assignation, Noeud (Name s, []) :: _)
      when not (Hashtbl.mem sub_intent s) ->
        Hashtbl.add sub_intent s Out
```

```
      | Noeud (Operateur Assignation, Noeud (Name s, []) :: _)
        when Hashtbl.find sub_intent s = In ->
          Hashtbl.replace sub_intent s InOut
      | Noeud (Operateur Assignation, Noeud (Name s, []) :: _)
        when Hashtbl.find sub_intent s = Out ||
          Hashtbl.find sub_intent s = InOut
        ->
          ()
      | Noeud (Name s, []) when not (Hashtbl.mem sub_intent s) ->
          Hashtbl.add sub_intent s In
      | Noeud (Name s, []) when Hashtbl.find sub_intent s = Out ->
          Hashtbl.add sub_intent s InOut
      | Noeud (Name s, [])
```

```
    when Hashtbl.find sub_intent s = Out || Hashtbl.find sub_intent s = InOut
    ->
      ()
  | Noeud (_, l) -> List.iter subroutine_intent_aux l

in
subroutine_intent_aux t;
sub_intent
```

```
type terminal =
    | EOF
    | E
    | PowerOp
    | NotOp
    | AndOp
    | OrOp
    | Dcon
    | Rcon
    | Icon
    | SconSingle
    | SconDouble
    | Ident
    | EOS
    | Return
    | Result
    | Contains
```

```
    | True
    | False
    | Program
    | Function
    | Subroutine
    | EndProgram
    | EndFunction
    | EndSubroutine
    | EndDo
    | EndIf
    | Colon
    | Comma
    | Equal
    | Asterisk
    | LParenthesis
    | RParenthesis
    | Integer
```

```
    | Real
    | Double
    | Complex
    | Character
    | Logical
    | Parameter
    | Intent
    | In
    | Out
    | InOut
    | Call
    | Print
    | Do
    | While
    | If
    | Else
    | Then
```

```
    | Divise
    | Plus
    | Minus
    | IsEqual
    | NotEqual
    | StrictLess
    | LessEqual
    | StrictGreater
    | GreaterEqual
    | Equivalent
    | NotEquivalent
    | Space
    | Recursive

type non_terminal =
    | ExecutableProgram
    | StartCommentBlock
```

```
    | Function_or_Subroutine_star_MainProgram
    | Function_or_Subroutine_star
    | Recursive_opt_Function_or_Subroutine
    | Function_or_Subroutine
    | MainProgram
    | MainRange
    | Contains_Function_opt_EndProgramStmt
    | Contains_Function
    | FunctionSubprogram_star
    | BodyConstruct_star
    | ProgramStmt
    | EndProgramStmt
    | FunctionSubprogram
    | FunctionPrefix
    | FunctionRange
    | FunctionParList
    | FunctionPar_Comma_FunctionPar_star_opt
```

```
    | Comma_FunctionPar_star
    | FunctionPar
    | FunctionResult_opt
    | EndFunctionStmt
    | SubroutineSubprogram
    | SubroutineRange
    | SubroutineParList_opt
    | SubroutinePar_Comma_SubroutinePar_star_opt
    | Comma_SubroutinePar_star
    | SubroutinePar
    | EndSubroutineStmt
    | EndName_opt
    | BodyConstruct
    | SpecificationPartConstruct
    | DeclarationConstruct
    | TypeDeclarationStmt
    | Comma_AttrSpec_star
```

```
    | AttrSpec
    | Intent_in_out
    | In_out
    | TypeDecl_Assignment
    | Comma_ObjectName_star
    | Comma_EntityDecl_star
    | EntityDecl
    | Equal_Expr_opt
    | Asterisk_CharLength_opt
    | CharLength
    | TypeParamValue
    | Expr_Or_Asterisk
    | TypeSpec
    | KindSelector_opt
    | ExecutableConstruct
    | ReturnStmt
    | ActionStmt
```

```
    | AssignmentStmt
    | PrintStmt
    | Comma_OutputItemList_opt
    | FormatIdentifier
    | OutputItemList
    | Comma_OutputItem_star
    | OutputItem
    | DoConstruct
    | BlockDoConstruct
    | LoopControl_opt
    | EndDoStmt
    | Name_opt
    | LoopControl
    | Comma_IntRealDpExpression_opt
    | IntRealDpExpression
    | IfConstruct
    | ElseIfStmt_ExecutionPartConstruct_star_star
```

```
    | ExecutionPartConstruct_star
    | ElseStmt_ExecutionPartConstruct_star_opt
    | IfThenStmt
    | ElseIfStmt
    | ElseStmt
    | EndIfStmt
    | ExecutionPartConstruct
    | ScalarLogicalExpr
    | Expr
    | Level5Expr
    | EquivOp_EquivOperand_star
    | EquivOperand
    | OrOp_OrOperand_star
    | OrOperand
    | AndOp_AndOperand_star
    | AndOperand
    | NotOp_opt
```

```
    | Level4Expr
    | RelOp_Level3Expr_star
    | Level3Expr
    | Level2Expr
    | AddOp_Sign_opt_AddOperand_star
    | Sign_opt_AddOperand
    | Sign_opt
    | AddOperand
    | MultOp_MultOperand_star
    | MultOperand
    | PowerOp_Level1Expr_star
    | Level1Expr
    | Primary
    | FunctionReference_opt
    |
    ↪  FunctionArg_Comma_FunctionArg_star_opt_RParenthes
    | Comma_FunctionArg_star
```

```
    | FunctionArg
    | Name
    | ArrayName
    | ComponentName
    | EndName
    | DummyArgName
    | FunctionName
    | ImpliedDoVariable
    | ProgramName
    | SubroutineName
    | SubroutineNameUse
    | VariableName
    | ObjectName
    | LogicalConstant
    | MultOp
    | AddOp
    | Sign
```

```
    | RelOp
    | EquivOp
    | ScalarIntLiteralConstant
    | Scon
type symbol = | Terminal of terminal | NonTerminal of non_terminal
let safe_token = Ident
let unparsed_tokens = [Space; ]

let repr_of_terminal (t : terminal) : string =
  match t with
    | EOF -> "End of file"
    | E -> "Epsilon"
    | PowerOp -> "\\*\\*"
    | NotOp -> "\\.not\\."
    | AndOp -> "\\.and\\."
    | OrOp -> "\\.or\\."
```

```
    | Dcon ->
    ↪ "(([0-9]+\\.[0-9]*)|(\\.[0-9]+))(d(\\+|-)?[0-9]+)
    | Rcon ->
    ↪ "(([0-9]+\\.[0-9]*)|(\\.[0-9]+))(e(\\+|-)?[0-9]+)
    | Icon -> "[0-9]+(e(\\+|-)?[0-9]+)?"
    | SconSingle -> "['](~[']|'')*[']"
    | SconDouble -> "[\"](~[\"]|\"\")*[\"]"
    | Ident -> "[A-Za-z][A-Za-z0-9_]*"
    | EOS -> "((!~[\\n]*)?\\n[ ]*)+"
    | Return -> "return"
    | Result -> "result"
    | Contains -> "contains"
    | True -> "\\.true\\."
    | False -> "\\.false\\."
    | Program -> "program"
    | Function -> "function"
    | Subroutine -> "subroutine"
```

```
    | EndProgram -> "end program"
    | EndFunction -> "end function"
    | EndSubroutine -> "end subroutine"
    | EndDo -> "end do"
    | EndIf -> "end if"
    | Colon -> ":"
    | Comma -> ","
    | Equal -> "="
    | Asterisk -> "\\*"
    | LParenthesis -> "\\("
    | RParenthesis -> "\\)"
    | Integer -> "integer"
    | Real -> "real"
    | Double -> "double precision"
    | Complex -> "complex"
    | Character -> "character"
    | Logical -> "logical"
```

```
    | Parameter -> "parameter"
    | Intent -> "intent"
    | In -> "in"
    | Out -> "out"
    | InOut -> "inout"
    | Call -> "call"
    | Print -> "print"
    | Do -> "do"
    | While -> "while"
    | If -> "if"
    | Else -> "else"
    | Then -> "then"
    | Divise -> "/"
    | Plus -> "\\+"
    | Minus -> "-"
    | IsEqual -> "(==)|(\\.eq\\.)"
    | NotEqual -> "(/=)|(\\.ne\\.)"
```

```
    | StrictLess -> "(<)|(\\.lt\\.)"
    | LessEqual -> "(<=)|(\\.le\\.)"
    | StrictGreater -> "(>)|(\\.gt\\.)"
    | GreaterEqual -> "(>=)|(\\.ge\\.)"
    | Equivalent -> "\\.eqv\\."
    | NotEquivalent -> "\\.neqv\\."
    | Space -> " "
    | Recursive -> "recursive"

let string_of_terminal (t : terminal) : string =
  match t with
    | EOF -> "EOF"
    | E -> "E"
    | PowerOp -> "PowerOp"
    | NotOp -> "NotOp"
    | AndOp -> "AndOp"
    | OrOp -> "OrOp"
```

```
    | Dcon -> "Dcon"
    | Rcon -> "Rcon"
    | Icon -> "Icon"
    | SconSingle -> "SconSingle"
    | SconDouble -> "SconDouble"
    | Ident -> "Ident"
    | EOS -> "EOS"
    | Return -> "Return"
    | Result -> "Result"
    | Contains -> "Contains"
    | True -> "True"
    | False -> "False"
    | Program -> "Program"
    | Function -> "Function"
    | Subroutine -> "Subroutine"
    | EndProgram -> "EndProgram"
    | EndFunction -> "EndFunction"
```

```
    | EndSubroutine -> "EndSubroutine"
    | EndDo -> "EndDo"
    | EndIf -> "EndIf"
    | Colon -> "Colon"
    | Comma -> "Comma"
    | Equal -> "Equal"
    | Asterisk -> "Asterisk"
    | LParenthesis -> "LParenthesis"
    | RParenthesis -> "RParenthesis"
    | Integer -> "Integer"
    | Real -> "Real"
    | Double -> "Double"
    | Complex -> "Complex"
    | Character -> "Character"
    | Logical -> "Logical"
    | Parameter -> "Parameter"
    | Intent -> "Intent"
```

```
    | In -> "In"
    | Out -> "Out"
    | InOut -> "InOut"
    | Call -> "Call"
    | Print -> "Print"
    | Do -> "Do"
    | While -> "While"
    | If -> "If"
    | Else -> "Else"
    | Then -> "Then"
    | Divise -> "Divise"
    | Plus -> "Plus"
    | Minus -> "Minus"
    | IsEqual -> "IsEqual"
    | NotEqual -> "NotEqual"
    | StrictLess -> "StrictLess"
    | LessEqual -> "LessEqual"
```

```
    | StrictGreater -> "StrictGreater"
    | GreaterEqual -> "GreaterEqual"
    | Equivalent -> "Equivalent"
    | NotEquivalent -> "NotEquivalent"
    | Space -> "Space"
    | Recursive -> "Recursive"

let string_of_non_terminal (nt : non_terminal) : string =
  match nt with
    | ExecutableProgram -> "ExecutableProgram"
    | StartCommentBlock -> "StartCommentBlock"
    | Function_or_Subroutine_star_MainProgram -> "Function_or_Subroutine_star_MainProgram"
    | Function_or_Subroutine_star -> "Function_or_Subroutine_star"
```

```
    | Recursive_opt_Function_or_Subroutine -> "Recursive_opt_Function_or_Subroutine"
    | Function_or_Subroutine -> "Function_or_Subroutine"
    | MainProgram -> "MainProgram"
    | MainRange -> "MainRange"
    | Contains_Function_opt_EndProgramStmt -> "Contains_Function_opt_EndProgramStmt"
    | Contains_Function -> "Contains_Function"
    | FunctionSubprogram_star -> "FunctionSubprogram_star"
    | BodyConstruct_star -> "BodyConstruct_star"
    | ProgramStmt -> "ProgramStmt"
    | EndProgramStmt -> "EndProgramStmt"
    | FunctionSubprogram -> "FunctionSubprogram"
    | FunctionPrefix -> "FunctionPrefix"
    | FunctionRange -> "FunctionRange"
```

```
    | FunctionParList -> "FunctionParList"
    | FunctionPar_Comma_FunctionPar_star_opt ->
    ↪  "FunctionPar_Comma_FunctionPar_star_opt"
    | Comma_FunctionPar_star ->
    ↪  "Comma_FunctionPar_star"
    | FunctionPar -> "FunctionPar"
    | FunctionResult_opt -> "FunctionResult_opt"
    | EndFunctionStmt -> "EndFunctionStmt"
    | SubroutineSubprogram -> "SubroutineSubprogram"
    | SubroutineRange -> "SubroutineRange"
    | SubroutineParList_opt ->
    ↪  "SubroutineParList_opt"
    | SubroutinePar_Comma_SubroutinePar_star_opt ->
    ↪  "SubroutinePar_Comma_SubroutinePar_star_opt"
    | Comma_SubroutinePar_star ->
    ↪  "Comma_SubroutinePar_star"
    | SubroutinePar -> "SubroutinePar"
```

```
    | EndSubroutineStmt -> "EndSubroutineStmt"
    | EndName_opt -> "EndName_opt"
    | BodyConstruct -> "BodyConstruct"
    | SpecificationPartConstruct ->
    ↪  "SpecificationPartConstruct"
    | DeclarationConstruct -> "DeclarationConstruct"
    | TypeDeclarationStmt -> "TypeDeclarationStmt"
    | Comma_AttrSpec_star -> "Comma_AttrSpec_star"
    | AttrSpec -> "AttrSpec"
    | Intent_in_out -> "Intent_in_out"
    | In_out -> "In_out"
    | TypeDecl_Assignment -> "TypeDecl_Assignment"
    | Comma_ObjectName_star ->
    ↪  "Comma_ObjectName_star"
    | Comma_EntityDecl_star ->
    ↪  "Comma_EntityDecl_star"
    | EntityDecl -> "EntityDecl"
```

```
    | Equal_Expr_opt -> "Equal_Expr_opt"
    | Asterisk_CharLength_opt ->
    ↪  "Asterisk_CharLength_opt"
    | CharLength -> "CharLength"
    | TypeParamValue -> "TypeParamValue"
    | Expr_Or_Asterisk -> "Expr_Or_Asterisk"
    | TypeSpec -> "TypeSpec"
    | KindSelector_opt -> "KindSelector_opt"
    | ExecutableConstruct -> "ExecutableConstruct"
    | ReturnStmt -> "ReturnStmt"
    | ActionStmt -> "ActionStmt"
    | AssignmentStmt -> "AssignmentStmt"
    | PrintStmt -> "PrintStmt"
    | Comma_OutputItemList_opt ->
    ↪  "Comma_OutputItemList_opt"
    | FormatIdentifier -> "FormatIdentifier"
    | OutputItemList -> "OutputItemList"
```

```
    | Comma_OutputItem_star ->
    ↪  "Comma_OutputItem_star"
    | OutputItem -> "OutputItem"
    | DoConstruct -> "DoConstruct"
    | BlockDoConstruct -> "BlockDoConstruct"
    | LoopControl_opt -> "LoopControl_opt"
    | EndDoStmt -> "EndDoStmt"
    | Name_opt -> "Name_opt"
    | LoopControl -> "LoopControl"
    | Comma_IntRealDpExpression_opt ->
    ↪  "Comma_IntRealDpExpression_opt"
    | IntRealDpExpression -> "IntRealDpExpression"
    | IfConstruct -> "IfConstruct"
    | ElseIfStmt_ExecutionPartConstruct_star_star ->
    ↪  "ElseIfStmt_ExecutionPartConstruct_star_star"
    | ExecutionPartConstruct_star ->
    ↪  "ExecutionPartConstruct_star"
```

```
    | ElseStmt_ExecutionPartConstruct_star_opt -> "ElseStmt_ExecutionPartConstruct_star_opt"
    | IfThenStmt -> "IfThenStmt"
    | ElseIfStmt -> "ElseIfStmt"
    | ElseStmt -> "ElseStmt"
    | EndIfStmt -> "EndIfStmt"
    | ExecutionPartConstruct -> "ExecutionPartConstruct"
    | ScalarLogicalExpr -> "ScalarLogicalExpr"
    | Expr -> "Expr"
    | Level5Expr -> "Level5Expr"
    | EquivOp_EquivOperand_star -> "EquivOp_EquivOperand_star"
    | EquivOperand -> "EquivOperand"
    | OrOp_OrOperand_star -> "OrOp_OrOperand_star"
    | OrOperand -> "OrOperand"
```

```
    | AndOp_AndOperand_star ->
    ↪  "AndOp_AndOperand_star"
    | AndOperand -> "AndOperand"
    | NotOp_opt -> "NotOp_opt"
    | Level4Expr -> "Level4Expr"
    | RelOp_Level3Expr_star ->
    ↪  "RelOp_Level3Expr_star"
    | Level3Expr -> "Level3Expr"
    | Level2Expr -> "Level2Expr"
    | AddOp_Sign_opt_AddOperand_star ->
    ↪  "AddOp_Sign_opt_AddOperand_star"
    | Sign_opt_AddOperand -> "Sign_opt_AddOperand"
    | Sign_opt -> "Sign_opt"
    | AddOperand -> "AddOperand"
    | MultOp_MultOperand_star ->
    ↪  "MultOp_MultOperand_star"
    | MultOperand -> "MultOperand"
```

```
| PowerOp_Level1Expr_star -> "PowerOp_Level1Expr_star"
| Level1Expr -> "Level1Expr"
| Primary -> "Primary"
| FunctionReference_opt -> "FunctionReference_opt"
| FunctionArg_Comma_FunctionArg_star_opt_RParenthes
  -> "FunctionArg_Comma_FunctionArg_star_opt_RParenthes
| Comma_FunctionArg_star -> "Comma_FunctionArg_star"
| FunctionArg -> "FunctionArg"
| Name -> "Name"
| ArrayName -> "ArrayName"
| ComponentName -> "ComponentName"
| EndName -> "EndName"
```

```
    | DummyArgName -> "DummyArgName"
    | FunctionName -> "FunctionName"
    | ImpliedDoVariable -> "ImpliedDoVariable"
    | ProgramName -> "ProgramName"
    | SubroutineName -> "SubroutineName"
    | SubroutineNameUse -> "SubroutineNameUse"
    | VariableName -> "VariableName"
    | ObjectName -> "ObjectName"
    | LogicalConstant -> "LogicalConstant"
    | MultOp -> "MultOp"
    | AddOp -> "AddOp"
    | Sign -> "Sign"
    | RelOp -> "RelOp"
    | EquivOp -> "EquivOp"
    | ScalarIntLiteralConstant -> "ScalarIntLiteralConstant"
    | Scon -> "Scon"
```

```
type string_or_string_list = S of string | L of string_or_string_list list

(** crée une chaîne de [n] tabulations *)
let tabs_to_string (n : int) : string_or_string_list = S (String.make n '\t')

(** crée une chaîne de [n] retours à la ligne *)
let rec n_new_lines (n : int) : string_or_string_list = S (String.make n '\n')


let string_of_string_or_string_list (sosl : string_or_string_list) : string =
  let rec aux (sosl : string_or_string_list) (acc : string list) : string list =
    match sosl with
```

```
      | S s -> s :: acc
      | L [] -> acc
      | L (e :: q) ->
          let acc2 = aux e acc in
          aux (L q) acc2
  in
  String.concat "" (List.rev (aux sosl []))


let rec last_of_list (l: 'a list): 'a =
  match l with
  | [] -> failwith "Liste vide "
  | e::[] -> e
  | e::q -> last_of_list q

let print_sosl (sosl: string_or_string_list): unit
 ↪  =
```

```
    print_string (string_of_string_or_string_list
     ↪  sosl);
    print_newline ()
```

```
open Symbols

type regex =
  (* cas de base *)
  | Epsilon
  | Caractere of char
  | AllChars
  | Range of char * char
  (* opérations sur les regex*)
  | Concat of regex * regex
  | Ou of regex * regex
  | UnPlus of regex
  | ZeroPlus of regex
  | Vide
  | Facultatif of regex
  | AllBut of bool array

```

```
(** affiche l'expression [c] en argument *)
let print_reg_list (c : regex list) : unit =
  (** affiche en vidant au fur et à mesure la
  ↪  liste *)
  let rec print_list_aux (r : regex list) : unit =
    match r with
    | [] -> ()
    | [ Vide ] -> print_char '_'
    | Vide :: q ->
        print_char '_';
        print_char ' ';
        print_list_aux q
    | [ Epsilon ] -> print_char '#'
    | Epsilon :: q ->
        print_char '#';
        print_char ' ';
        print_list_aux q
```

```
    | [ Caractere c ] -> print_char c
    | Caractere c :: q ->
      print_char c;
      print_char ' ';
      print_list_aux q
    | [ AllChars ] -> print_char '.'
    | AllChars :: q ->
      print_char '.';
      print_char ' ';
      print_list_aux q
    | [ Range (s, e) ] ->
      print_char '[';
      print_char s;
      print_char '-';
      print_char e;
      print_char ']'
    | Range (s, e) :: q ->
```

```
          print_char '[';
          print_char s;
          print_char '-';
          print_char e;
          print_char ']';
          print_char ' ';
          print_list_aux q
    | [ Concat (e1, e2) ] ->
          print_list_aux [ e1 ];
          print_list_aux [ e2 ]
    | Concat (e1, e2) :: q ->
          print_list_aux [ e1 ];
          print_list_aux [ e2 ];
          print_char ' ';
          print_list_aux q
    | [ Ou (e1, e2) ] ->
          print_char '(';
```

```
            print_list_aux [ e1 ];
            print_char '|';
            print_list_aux [ e2 ];
            print_char ')'
        | Ou (e1, e2) :: q ->
            print_char '(';
            print_list_aux [ e1 ];
            print_char '|';
            print_list_aux [ e2 ];
            print_char ')';
            print_char ' ';
            print_list_aux q
        | [ UnPlus e ] ->
            print_char '(';
            print_list_aux [ e ];
            print_string ")+"
        | UnPlus e :: q ->
```

```
        print_char '(';
        print_list_aux [ e ];
        print_string ")+";
        print_char ' ';
        print_list_aux q
    | [ ZeroPlus e ] ->
        print_char '(';
        print_list_aux [ e ];
        print_string ")*"
    | ZeroPlus e :: q ->
        print_char '(';
        print_list_aux [ e ];
        print_string ")*";
        print_char ' ';
        print_list_aux q
    | [ Facultatif e ] ->
        print_char '(';
```

```
          print_list_aux [ e ];
          print_string ")?"
      | Facultatif e :: q ->
          print_char '(';
          print_list_aux [ e ];
          print_string ")?";
          print_char ' ';
          print_list_aux q
      | [ AllBut _ ] ->
          print_string "~(...";
          print_string ")"
      | AllBut _ :: q ->
          print_string "~(...";
          print_string ")";
          print_char ' ';
          print_list_aux q
  in
```

```
  if List.length c = 0 then print_string "[]" else print_list_aux c

(** convertit la chaîne de caractères s à partir de l'index index et l'ajoute à
    la liste c *)
let rec string_to_char_2 (s : string) (c : char list) (index : int) : char list
    =
  if index = String.length s then List.rev c
  else string_to_char_2 s (s.[index] :: c) (index + 1)

exception Invalid_syntax
exception Empty_pile

```

```
(** teste si la liste référencée dans [l] est vide
↪  *)
let is_empty (l : 'a list ref) : bool = List.length
↪  !l == 0

(** dépile un élément de la pile référencée par
↪  [l] *)
let pop (l : 'a list ref) : 'a =
  match !l with
  | [] -> raise Empty_pile
  | x :: q ->
      l := q;
      x

(** convertit l'entier [n] en booléen *)
let bool_of_int (n : int) : bool = if n == 0 then
↪  false else true
```

```ocaml

(** renvoie la disjonction de toutes les
↪  expressions régulières dans la liste
   [l] *)
let or_reg (l : regex list) : regex =
  let rec or_reg_aux (l : regex list) (out : regex)
  ↪  : regex =
    match (l, out) with
    | [], _ -> out
    | x :: q, Epsilon -> or_reg_aux q x
    | x :: q, _ -> or_reg_aux q (Ou (x, out))
  in
  or_reg_aux l Epsilon

(** renvoie la concaténation de toutes les
↪  expressions régulières de [l] *)
let concat_reg (l : regex list) : regex =
```

```
  let rec concat_reg_aux (l : regex list) (out : regex) : regex =
    match (l, out) with
    | [], _ -> out
    | x :: q, Epsilon -> concat_reg_aux q x
    | x :: q, _ -> concat_reg_aux q (Concat (x, out))
  in
  concat_reg_aux l Epsilon

(** transforme, si c'est possible, la chaine de caractères [s] en une expression
    régulière (regex) *)
let rec gen_regex (s : string) : regex =
  let caracters = ref (List.of_seq (String.to_seq s)) in
```

```ocaml
(** fonction auxiliaire qui permet de générer le regex entre parenthèses *)
let parenthesis () : regex =
  let l = ref [] in
  try
    let count = ref 0 in
    let c = ref (pop caracters) in
    let ignore = ref false in
    (* boucle sur le contenu de la parenthèse et ne s'arrete pas si elle est ignorée *)
    while not (!c = ')' && !count = 0 && not !ignore) do
      if not !ignore then (
        if
          (* si on rencontre un \, on ignore le caractère suivant *)
          !c = '\\'
```

```
          then ignore := true
          else if !c = '(' then count := !count + 1
          else if !c = ')' then count := !count - 1)
        else ignore := false;
        l := !c :: !l;
        c := pop caracters
      done;
      let s1 = String.of_seq (List.to_seq (List.rev !l)) in
      gen_regex s1
    with Empty_pile -> raise Invalid_syntax
  in

  (** fonction auxiliaire qui permet de générer le regex entre crochets *)
  let crochet () : regex =
```

```
    let l = ref [] in
    try
      let c = ref (pop caracters) in
      let ignore_left = ref 0 in
      let pile1 = ref [] in
      (* on boucle tant que l'on a pas le crochet final et tant que l'on ignore pas *)
      while not (!c = ']' && not (bool_of_int !ignore_left)) do
        (* si on ignore pas encore, on ignore pour 2 tours si on rencontre un \ *)
        if (not (bool_of_int !ignore_left)) && !c = '\\' then ignore_left := 2;
        (if !ignore_left <> 2 then
           match !pile1 with
           | [ Caractere '-'; Caractere x ] -> pile1 := [ Range (x, !c) ]
```

```
              | Caractere '-' :: Caractere x :: q -> pile1 := Range (x, !c) :: q
              | _ ->
                 if !ignore_left = 1 && !c = 'n' then
                   pile1 := Caractere '\n' :: !pile1
                 else pile1 := Caractere !c :: !pile1;
                 l := !c :: !l);
          c := pop caracters;
          if !ignore_left > 0 then ignore_left := !ignore_left - 1
        done;
        or_reg !pile1
      with Empty_pile -> raise Invalid_syntax
    in

```

```
(** fonction auxiliaire générale qui génère le regex à partir de la chaine,
    qui stocke au fur et à mesure dans la pile et qui ignore les caractères à
    effets lorsque ignore est à vrai *)
let rec gen_regex_2 (pile : regex list) (ignore : bool) : regex =
  (* s'il n'y a plus rien a convertir, on concatène les expressions régulières*)
  if is_empty caracters then concat_reg pile
  else
    let c = pop caracters in
    if ignore then
      if c = 'n' then gen_regex_2 (Caractere '\n' :: pile) false
      else gen_regex_2 (Caractere c :: pile) false
```

```
        else
          match c with
           | '\\' -> gen_regex_2 pile true
           | '(' -> gen_regex_2 (parenthesis () :: pile) false
           | '[' -> gen_regex_2 (crochet () :: pile) false
           | ')' | ']' -> raise Invalid_syntax
           | '|' -> (
               let left = gen_regex_2 [] false in
               match (pile, left) with
               | [], _ -> raise Invalid_syntax
               | [ Caractere c ], AllChars | [ AllChars ], Caractere c ->
                   if c = '\n' then
                     gen_regex_2 [ Ou (Caractere c, AllChars) ] false
```

```
      else gen_regex_2 [ AllChars ] false
  | Caractere c :: q, AllChars | AllChars :: q, Caractere c ->
      if c = '\n' then
        gen_regex_2 (Ou (Caractere c, AllChars) :: q) false
      else gen_regex_2 (AllChars :: q) false
  | [ Range (d, f) ], AllChars | [ AllChars ], Range (d, f) ->
      if int_of_char d < 32 || int_of_char f < 32 then
        gen_regex_2 [ Ou (Range (d, f), AllChars) ] false
      else gen_regex_2 [ AllChars ] false
  | Range (d, f) :: q, AllChars | AllChars :: q, Range (d, f) ->
```

```
                if int_of_char d < 32 || int_of_char f < 32 then
                  gen_regex_2 (Ou (Range (d, f), AllChars) :: q) false
                else gen_regex_2 (AllChars :: q) false
            | [ x ], _ -> gen_regex_2 [ Ou (x, left) ] false
            | x :: q, _ -> gen_regex_2 (Ou (x, left) :: q) false)
        | '#' -> (
            match pile with
            | [] -> gen_regex_2 [ Epsilon ] false
            | Concat (_, _) :: _
            | Caractere _ :: _
            | Ou (_, _) :: _
            | Range (_, _) :: _ ->
```

```
                gen_regex_2 pile false
            | _ -> gen_regex_2 (Epsilon :: pile) false)
        | '+' -> (
            match pile with
            | [] -> raise Invalid_syntax
            | Epsilon :: q -> gen_regex_2 pile false
            | [ x ] -> gen_regex_2 [ UnPlus x ] false
            | x :: q -> gen_regex_2 (UnPlus x :: q) false)
        | '*' -> (
            match pile with
            | [] -> raise Invalid_syntax
            | Epsilon :: q -> gen_regex_2 pile false
```

```
                    | [ x ] -> gen_regex_2 [ ZeroPlus x ] false
                    | x :: q -> gen_regex_2 (ZeroPlus x :: q) false)
                | '.' -> gen_regex_2 (AllChars :: pile) false
                | '?' -> (
                    match pile with
                    | [] -> raise Invalid_syntax
                    | Epsilon :: q -> gen_regex_2 pile false
                    | [ x ] -> gen_regex_2 [ Facultatif x ] false
                    | x :: q -> gen_regex_2 (Facultatif x :: q) false)
                | '~' -> (
                    let c = ref (pop caracters) in
```

```
                match !c with
                | '[' ->
                    let reg = crochet () in
                    let rec recon_crochet e =
                      match e with
                      | Caractere c ->
                          gen_regex_2
                            (AllBut (Array.init 128 (( <> ) (int_of_char c)))
                            :: pile)
                            false
                      | Range (b1, b2) ->
                          gen_regex_2
                            (AllBut
                               (Array.init 128 (fun i ->
```

```
                                    i < int_of_char b1 || i > int_of_char b2))
                              :: pile)
                              false
                    | Ou (e1, e2) -> (
                        match (recon_crochet e1, recon_crochet e2) with
                        | AllBut l1, AllBut l2 ->
                            gen_regex_2
                              (AllBut (Array.map2 ( && ) l1 l2) :: pile)
                              false
                        | _ -> failwith "impossible")
                    | _ -> raise Invalid_syntax
```

```
                    in
                    recon_crochet reg
                | '(' ->
                    let reg = parenthesis () in
                    let rec recon_parntethis e =
                      match e with
                      | Caractere c ->
                          gen_regex_2
                            (AllBut (Array.init 128 ((
                            ↪  <> ) (int_of_char c)))
                            :: pile)
                            false
                      | Range (b1, b2) ->
                          gen_regex_2
                            (AllBut
                               (Array.init 128 (fun i
                               ↪  ->
```

```
                                    i < int_of_char b1 || i > int_of_char b2))
                              :: pile)
                              false
                       | AllChars ->
                            gen_regex_2
                              (AllBut (Array.init 128 (( < ) 32)) :: pile)
                              false
                       | Ou (e1, e2) -> (
                            match (recon_parntethis e1, recon_parntethis e2) with
                            | AllBut l1, AllBut l2 ->
                                gen_regex_2
```

```
                                (AllBut (Array.map2 ( && ) l1 l2) :: pile)
                                false
                          | _ -> failwith "impossible")
                      | _ -> raise Invalid_syntax
                  in
                  recon_parntethis reg
              | '\\' ->
                  gen_regex_2
                    (AllBut
                       (Array.init 128 (( <> ) (int_of_char (pop caracters))))
                    :: pile)
                    false
              | ')' | ']' -> raise Invalid_syntax
```

```
                | c1 ->
                    gen_regex_2
                      (AllBut (Array.init 128 (( <> ) (int_of_char c1))) :: pile)
                      false)
            | _ -> gen_regex_2 (Caractere c :: pile) false
  in
  let reg = gen_regex_2 [] false in
  if not (is_empty caracters) then raise Invalid_syntax else reg
```

```ocaml
open Symbols

type pattern = symbol list
type rule = symbol * pattern list
type rule_hashtable = (symbol, pattern list)
↪ Hashtbl.t
type grammar = { rules_htbl : rule_hashtable;
↪ start_symbol : Symbols.symbol }

(** pour un symbole [s], renvoie sa représentation
↪ sous forme de chaîne *)
let string_of_symbol (s : symbol) : string =
  match s with
  | Terminal t -> string_of_terminal t
  | NonTerminal nt -> string_of_non_terminal nt

(** affiche le symbole [s] *)
```

```ocaml
let print_symbol (s : symbol) : unit = print_string (string_of_symbol s)

(** affiche chaque pattern de [patterns] *)
let print_patterns (patterns : pattern list) : unit =
  List.iter
    (fun pattern ->
      List.iter (fun s -> print_string (string_of_symbol s ^ " ")) pattern;
      print_newline ())
    patterns

(** affiche chaque règle de [r] *)
let print_rule_list (r : rule_hashtable) : unit =
  Hashtbl.iter
    (fun s patterns ->
```

```
        print_string (string_of_symbol s ^ " -> \n");
        print_patterns patterns;
        print_newline ())
    r

(** affiche la grammaire [g] *)
let print_grammar (g : grammar) =
  print_string "{\nrules: ";
  print_rule_list g.rules_htbl;
  print_string "\nstart_symbol: ";
  print_symbol g.start_symbol;
  print_string "\n}\n"

(** teste si le symbole [s] est terminal *)
let is_terminal_symbol (s : symbol) : bool =
  match s with Terminal _ -> true | NonTerminal _
   ↪  -> false
```

```ocaml

(** teste si le symbole [s] est non terminal *)
let is_non_terminal_symbol (s : symbol) : bool = not (is_terminal_symbol s)

(** teste si la règle associe un symbole [s] est terminal *)
let is_terminal ((s, _) : rule) : bool = is_terminal_symbol s

(** teste si la règle [r] est non terminale*)
let is_non_terminal (r : rule) : bool = not (is_terminal r)

(** renvoie la liste des terminaux de [r] *)
let terminals (r : rule list) : rule list = List.filter is_terminal r
```

```ocaml

(** renvoie la liste des non terminaux de [r] *)
let non_terminals (r : rule list) : rule list = List.filter is_non_terminal r

(** renvoie la règle associée au symbole [s] dans la grammaire [g] *)
let rule_of_symbol (g : grammar) (s : symbol) : rule =
  if Hashtbl.mem g.rules_htbl s then (s, Hashtbl.find g.rules_htbl s)
  else
    failwith
      ("the hashed grammar does not contain " ^ string_of_symbol s
      ^ " as a key\n")
```

```
open Grammar_functions
open Symbols
let grammar = { start_symbol = NonTerminal
 ↪  ExecutableProgram;
rules_htbl = Hashtbl.of_seq (List.to_seq
 ↪  [(NonTerminal ExecutableProgram,[[NonTerminal
 ↪  StartCommentBlock;NonTerminal
 ↪  Function_or_Subroutine_star_MainProgram;];[NonTermi
 ↪  Function_or_Subroutine_star_MainProgram;];]);
(NonTerminal StartCommentBlock,[[Terminal EOS;];]);
(NonTerminal
 ↪  Function_or_Subroutine_star_MainProgram,[[NonTermina
 ↪  Recursive_opt_Function_or_Subroutine;NonTerminal
 ↪  Function_or_Subroutine_star_MainProgram;];[NonTermi
 ↪  MainProgram;NonTerminal
 ↪  Function_or_Subroutine_star;];]);
```

```
(NonTerminal Function_or_Subroutine_star,[[NonTerminal Recursive_opt_Function_or_Subroutine;NonTerminal Function_or_Subroutine_star;];[Terminal E;];]);
(NonTerminal Recursive_opt_Function_or_Subroutine,[[Terminal Recursive;NonTerminal Function_or_Subroutine;];[NonTerminal Function_or_Subroutine;];]);
(NonTerminal Function_or_Subroutine,[[NonTerminal FunctionSubprogram;];[NonTerminal SubroutineSubprogram;];]);
(NonTerminal MainProgram,[[NonTerminal ProgramStmt;NonTerminal MainRange;];]);
```

```
(NonTerminal MainRange,[[NonTerminal BodyConstruct;NonTerminal BodyConstruct_star;NonTerminal Contains_Function_opt_EndProgramStmt;];[NonTerminal Contains_Function_opt_EndProgramStmt;];]);
(NonTerminal Contains_Function_opt_EndProgramStmt,[[NonTerminal Contains_Function;NonTerminal EndProgramStmt;];[NonTerminal EndProgramStmt;];]);
(NonTerminal Contains_Function,[[Terminal Contains;Terminal EOS;NonTerminal FunctionSubprogram_star;];]);
```

```
(NonTerminal FunctionSubprogram_star,[[Terminal Recursive;NonTerminal FunctionSubprogram;NonTerminal FunctionSubprogram_star;];[NonTerminal FunctionSubprogram;NonTerminal FunctionSubprogram_star;];[Terminal E;];]);
(NonTerminal BodyConstruct_star,[[NonTerminal BodyConstruct;NonTerminal BodyConstruct_star;];[Terminal E;];]);
(NonTerminal ProgramStmt,[[Terminal Program;NonTerminal ProgramName;Terminal EOS;];]);
(NonTerminal EndProgramStmt,[[Terminal EndProgram;NonTerminal EndName_opt;Terminal EOS;];]);
```

```
(NonTerminal FunctionSubprogram,[[NonTerminal FunctionPrefix;NonTerminal FunctionName;NonTerminal FunctionRange;];]);
(NonTerminal FunctionPrefix,[[NonTerminal TypeSpec;Terminal Function;];[Terminal Function;];]);
(NonTerminal FunctionRange,[[NonTerminal FunctionParList;NonTerminal FunctionResult_opt;Terminal EOS;NonTerminal BodyConstruct_star;NonTerminal EndFunctionStmt;];]);
(NonTerminal FunctionParList,[[Terminal LParenthesis;NonTerminal FunctionPar_Comma_FunctionPar_star_opt;Terminal RParenthesis;];]);
```

```
(NonTerminal FunctionPar_Comma_FunctionPar_star_opt,[[NonTerminal FunctionPar;NonTerminal Comma_FunctionPar_star;];[Terminal E;];]);
(NonTerminal Comma_FunctionPar_star,[[Terminal Comma;NonTerminal FunctionPar;NonTerminal Comma_FunctionPar_star;];[Terminal E;];]);
(NonTerminal FunctionPar,[[NonTerminal DummyArgName;];]);
(NonTerminal FunctionResult_opt,[[Terminal Result;Terminal LParenthesis;NonTerminal VariableName;Terminal RParenthesis;];[Terminal E;];]);
(NonTerminal EndFunctionStmt,[[Terminal EndFunction;NonTerminal EndName_opt;Terminal EOS;];]);
```

```
(NonTerminal SubroutineSubprogram,[[Terminal Subroutine;NonTerminal SubroutineName;NonTerminal SubroutineRange;];]);
(NonTerminal SubroutineRange,[[NonTerminal SubroutineParList_opt;Terminal EOS;NonTerminal BodyConstruct_star;NonTerminal EndSubroutineStmt;];]);
(NonTerminal SubroutineParList_opt,[[Terminal LParenthesis;NonTerminal SubroutinePar_Comma_SubroutinePar_star_opt;Terminal RParenthesis;];[Terminal E;];]);
(NonTerminal SubroutinePar_Comma_SubroutinePar_star_opt,[[NonTerm
SubroutinePar;NonTerminal Comma_SubroutinePar_star;];[Terminal E;];]);
```

```
(NonTerminal Comma_SubroutinePar_star,[[Terminal Comma;NonTerminal SubroutinePar;NonTerminal Comma_SubroutinePar_star;];[Terminal E;];]);
(NonTerminal SubroutinePar,[[NonTerminal DummyArgName;];]);
(NonTerminal EndSubroutineStmt,[[Terminal EndSubroutine;NonTerminal EndName_opt;Terminal EOS;];]);
(NonTerminal EndName_opt,[[NonTerminal EndName;];[Terminal E;];]);
(NonTerminal BodyConstruct,[[NonTerminal SpecificationPartConstruct;];[NonTerminal ExecutableConstruct;];]);
(NonTerminal SpecificationPartConstruct,[[NonTerminal DeclarationConstruct;];]);
```

```
(NonTerminal DeclarationConstruct,[[NonTerminal TypeDeclarationStmt;];]);
(NonTerminal TypeDeclarationStmt,[[NonTerminal TypeSpec;NonTerminal Comma_AttrSpec_star;NonTerminal TypeDecl_Assignment;Terminal EOS;];]);
(NonTerminal Comma_AttrSpec_star,[[Terminal Comma;NonTerminal AttrSpec;NonTerminal Comma_AttrSpec_star;];[Terminal E;];]);
(NonTerminal AttrSpec,[[Terminal Parameter;];[NonTerminal Intent_in_out;];]);
(NonTerminal Intent_in_out,[[Terminal Intent;Terminal LParenthesis;NonTerminal In_out;Terminal RParenthesis;];]);
(NonTerminal In_out,[[Terminal In;];[Terminal Out;];[Terminal InOut;];]);
```

```
(NonTerminal TypeDecl_Assignment,[[Terminal Colon;Terminal Colon;NonTerminal EntityDecl;NonTerminal Comma_EntityDecl_star;];[NonTerminal ObjectName;NonTerminal Comma_ObjectName_star;];]);
(NonTerminal Comma_ObjectName_star,[[Terminal Comma;NonTerminal ObjectName;NonTerminal Comma_ObjectName_star;];[Terminal E;];]);
(NonTerminal Comma_EntityDecl_star,[[Terminal Comma;NonTerminal EntityDecl;NonTerminal Comma_EntityDecl_star;];[Terminal E;];]);
(NonTerminal EntityDecl,[[NonTerminal ObjectName;NonTerminal Asterisk_CharLength_opt;NonTerminal Equal_Expr_opt;];]);
```

```
(NonTerminal Equal_Expr_opt,[[Terminal Equal;NonTerminal Expr;];[Terminal E;];]);
(NonTerminal Asterisk_CharLength_opt,[[Terminal Asterisk;NonTerminal CharLength;];[Terminal E;];]);
(NonTerminal CharLength,[[Terminal LParenthesis;NonTerminal TypeParamValue;Terminal RParenthesis;];[NonTerminal ScalarIntLiteralConstant;];]);
(NonTerminal TypeParamValue,[[NonTerminal Expr_Or_Asterisk;];]);
(NonTerminal Expr_Or_Asterisk,[[NonTerminal Expr;];[Terminal Asterisk;];]);
```

```
(NonTerminal TypeSpec,[[Terminal Integer;NonTerminal KindSelector_opt;];[Terminal Double;];[Terminal Complex;NonTerminal KindSelector_opt;];[Terminal Logical;NonTerminal KindSelector_opt;];[Terminal Real;NonTerminal KindSelector_opt;];[Terminal Character;];]);
(NonTerminal KindSelector_opt,[[Terminal LParenthesis;NonTerminal Expr;Terminal RParenthesis;];[Terminal E;];]);
(NonTerminal ExecutableConstruct,[[NonTerminal ActionStmt;];[NonTerminal DoConstruct;];[NonTerminal IfConstruct;];[NonTerminal ReturnStmt;];]);
(NonTerminal ReturnStmt,[[Terminal Return;Terminal EOS;];]);
```

```
(NonTerminal ActionStmt,[[NonTerminal AssignmentStmt;];[NonTerminal PrintStmt;];]);
(NonTerminal AssignmentStmt,[[NonTerminal Name;Terminal Equal;NonTerminal Expr;Terminal EOS;];]);
(NonTerminal PrintStmt,[[Terminal Print;NonTerminal FormatIdentifier;NonTerminal Comma_OutputItemList_opt;Terminal EOS;];]);
(NonTerminal Comma_OutputItemList_opt,[[Terminal Comma;NonTerminal OutputItemList;];[Terminal E;];]);
(NonTerminal FormatIdentifier,[[Terminal Asterisk;];]);
(NonTerminal OutputItemList,[[NonTerminal OutputItem;NonTerminal Comma_OutputItem_star;];]);
```

```
(NonTerminal Comma_OutputItem_star,[[Terminal Comma;NonTerminal OutputItem;NonTerminal Comma_OutputItem_star;];[Terminal E;];]);
(NonTerminal OutputItem,[[NonTerminal Expr;];]);
(NonTerminal DoConstruct,[[NonTerminal BlockDoConstruct;];]);
(NonTerminal BlockDoConstruct,[[Terminal Do;NonTerminal LoopControl_opt;NonTerminal ExecutionPartConstruct_star;NonTerminal EndDoStmt;];]);
(NonTerminal LoopControl_opt,[[NonTerminal LoopControl;Terminal EOS;];[Terminal EOS;];]);
(NonTerminal EndDoStmt,[[Terminal EndDo;NonTerminal Name_opt;Terminal EOS;];]);
(NonTerminal Name_opt,[[NonTerminal Name;];[Terminal E;];]);
```

```
(NonTerminal LoopControl,[[Terminal While;Terminal
↪   LParenthesis;NonTerminal Expr;Terminal
↪   RParenthesis;];[NonTerminal
↪   VariableName;Terminal Equal;NonTerminal
↪   IntRealDpExpression;Terminal Comma;NonTerminal
↪   IntRealDpExpression;NonTerminal
↪   Comma_IntRealDpExpression_opt;];]);
(NonTerminal
↪   Comma_IntRealDpExpression_opt,[[Terminal
↪   Comma;NonTerminal
↪   IntRealDpExpression;];[Terminal E;];]);
(NonTerminal IntRealDpExpression,[[NonTerminal
↪   Expr;];]);
```

```
(NonTerminal IfConstruct,[[NonTerminal
↪   IfThenStmt;NonTerminal
↪   ExecutionPartConstruct_star;NonTerminal
↪   ElseIfStmt_ExecutionPartConstruct_star_star;NonTerm
↪   ElseStmt_ExecutionPartConstruct_star_opt;NonTermina
↪   EndIfStmt;];]);
(NonTerminal
↪   ElseIfStmt_ExecutionPartConstruct_star_star,[[NonTe
↪   ElseIfStmt;NonTerminal
↪   ExecutionPartConstruct_star;NonTerminal
↪   ElseIfStmt_ExecutionPartConstruct_star_star;];[Term
↪   E;];]);
(NonTerminal
↪   ExecutionPartConstruct_star,[[NonTerminal
↪   ExecutionPartConstruct;NonTerminal
↪   ExecutionPartConstruct_star;];[Terminal E;];]);
```

```
(NonTerminal
↪   ElseStmt_ExecutionPartConstruct_star_opt,[[NonTermi
↪   ElseStmt;NonTerminal
↪   ExecutionPartConstruct_star;];[Terminal E;];]);
(NonTerminal IfThenStmt,[[Terminal If;Terminal
↪   LParenthesis;NonTerminal
↪   ScalarLogicalExpr;Terminal
↪   RParenthesis;Terminal Then;Terminal EOS;];]);
(NonTerminal ElseIfStmt,[[Terminal Else;Terminal
↪   If;Terminal LParenthesis;NonTerminal
↪   ScalarLogicalExpr;Terminal
↪   RParenthesis;Terminal Then;Terminal EOS;];]);
(NonTerminal ElseStmt,[[Terminal Else;Terminal
↪   EOS;];]);
(NonTerminal EndIfStmt,[[Terminal EndIf;Terminal
↪   EOS;];]);
```

```
(NonTerminal ExecutionPartConstruct,[[NonTerminal ExecutableConstruct;];]);
(NonTerminal ScalarLogicalExpr,[[NonTerminal Expr;];]);
(NonTerminal Expr,[[NonTerminal Level5Expr;];]);
(NonTerminal Level5Expr,[[NonTerminal EquivOperand;NonTerminal EquivOp_EquivOperand_star;];]);
(NonTerminal EquivOp_EquivOperand_star,[[NonTerminal EquivOp;NonTerminal EquivOperand;NonTerminal EquivOp_EquivOperand_star;];[Terminal E;];]);
(NonTerminal EquivOperand,[[NonTerminal OrOperand;NonTerminal OrOp_OrOperand_star;];]);
(NonTerminal OrOp_OrOperand_star,[[Terminal OrOp;NonTerminal OrOperand;NonTerminal OrOp_OrOperand_star;];[Terminal E;];]);
```

```
(NonTerminal OrOperand,[[NonTerminal AndOperand;NonTerminal AndOp_AndOperand_star;];]);
(NonTerminal AndOp_AndOperand_star,[[Terminal AndOp;NonTerminal AndOperand;NonTerminal AndOp_AndOperand_star;];[Terminal E;];]);
(NonTerminal AndOperand,[[NonTerminal NotOp_opt;NonTerminal Level4Expr;];]);
(NonTerminal NotOp_opt,[[Terminal NotOp;];[Terminal E;];]);
(NonTerminal Level4Expr,[[NonTerminal Level3Expr;NonTerminal RelOp_Level3Expr_star;];]);
(NonTerminal RelOp_Level3Expr_star,[[NonTerminal RelOp;NonTerminal Level3Expr;NonTerminal RelOp_Level3Expr_star;];[Terminal E;];]);
```

```
(NonTerminal Level3Expr,[[NonTerminal Level2Expr;];]);
(NonTerminal Level2Expr,[[NonTerminal Sign_opt_AddOperand;NonTerminal AddOp_Sign_opt_AddOperand_star;];]);
(NonTerminal AddOp_Sign_opt_AddOperand_star,[[NonTerminal AddOp;NonTerminal Sign_opt_AddOperand;NonTerminal AddOp_Sign_opt_AddOperand_star;];[Terminal E;];]);
(NonTerminal Sign_opt_AddOperand,[[NonTerminal Sign_opt;NonTerminal AddOperand;];]);
(NonTerminal Sign_opt,[[NonTerminal Sign;];[Terminal E;];]);
```

```
(NonTerminal AddOperand,[[NonTerminal MultOperand;NonTerminal MultOp_MultOperand_star;];]);
(NonTerminal MultOp_MultOperand_star,[[NonTerminal MultOp;NonTerminal MultOperand;NonTerminal MultOp_MultOperand_star;];[Terminal E;];]);
(NonTerminal MultOperand,[[NonTerminal Level1Expr;NonTerminal PowerOp_Level1Expr_star;];]);
(NonTerminal PowerOp_Level1Expr_star,[[Terminal PowerOp;NonTerminal Level1Expr;NonTerminal PowerOp_Level1Expr_star;];[Terminal E;];]);
(NonTerminal Level1Expr,[[NonTerminal Primary;];]);
```

```
(NonTerminal Primary,[[Terminal Icon;];[Terminal
    Rcon;];[Terminal Dcon;];[NonTerminal
    Name;NonTerminal
    FunctionReference_opt;];[NonTerminal
    Scon;];[NonTerminal LogicalConstant;];[Terminal
    LParenthesis;NonTerminal Expr;Terminal
    RParenthesis;];]);
(NonTerminal FunctionReference_opt,[[Terminal
    LParenthesis;NonTerminal
    FunctionArg_Comma_FunctionArg_star_opt_RParenthesis
    E;];]);
(NonTerminal
    FunctionArg_Comma_FunctionArg_star_opt_RParenthesis
    FunctionArg;NonTerminal
    Comma_FunctionArg_star;Terminal
    RParenthesis;];[Terminal RParenthesis;];]);
```

```
(NonTerminal Comma_FunctionArg_star,[[Terminal Comma;NonTerminal FunctionArg;NonTerminal Comma_FunctionArg_star;];[Terminal E;];]);
(NonTerminal FunctionArg,[[NonTerminal Expr;];]);
(NonTerminal Name,[[Terminal Ident;];]);
(NonTerminal ArrayName,[[Terminal Ident;];]);
(NonTerminal ComponentName,[[Terminal Ident;];]);
(NonTerminal EndName,[[Terminal Ident;];]);
(NonTerminal DummyArgName,[[Terminal Ident;];]);
(NonTerminal FunctionName,[[Terminal Ident;];]);
(NonTerminal ImpliedDoVariable,[[Terminal Ident;];]);
(NonTerminal ProgramName,[[Terminal Ident;];]);
(NonTerminal SubroutineName,[[Terminal Ident;];]);
(NonTerminal SubroutineNameUse,[[Terminal Ident;];]);
(NonTerminal VariableName,[[Terminal Ident;];]);
```

```
(NonTerminal ObjectName,[[Terminal Ident;];]);
(NonTerminal LogicalConstant,[[Terminal True;];[Terminal False;];]);
(NonTerminal MultOp,[[Terminal Asterisk;];[Terminal Divise;];]);
(NonTerminal AddOp,[[NonTerminal Sign;];]);
(NonTerminal Sign,[[Terminal Plus;];[Terminal Minus;];]);
(NonTerminal RelOp,[[Terminal IsEqual;];[Terminal NotEqual;];[Terminal StrictLess;];[Terminal LessEqual;];[Terminal StrictGreater;];[Terminal GreaterEqual;];]);
(NonTerminal EquivOp,[[Terminal Equivalent;];[Terminal NotEquivalent;];]);
(NonTerminal ScalarIntLiteralConstant,[[Terminal Icon;];]);
```

```
(NonTerminal Scon,[[Terminal SconSingle;];[Terminal
↪  SconDouble;];]);
]))}
```

```
open Regex
open Symbols
open Vector
module IntSet = Set.Make (Int)

(* différentes représentations d'automates utiles lors de sa transformation en automate déterministe *)
type automate = {
  nodes : int list;
  debut_l : int list;
  fin : (int * terminal) list;
  transitions : (char option * int) list array;
}

type automate_sans_eps = {
  nodes : int list;
```

```
  debut_l : int list;
  fin : (int * terminal) list;
  transitions_sans_eps : (char * int) list array;
}

type pre_automate_det = {
  mutable nodes : int list;
  debut : int;
  mutable fin : terminal option array;
  mutable pre_transitions : int array Vector.t;
}

(* automate de sortie utile pour la transpilation
 ↪  *)
type automate_det = {
  nodes : int list;
  debut : int;
```

```
  fin : terminal option array;
  transitions : int array array;
      (* transitions.(i).(j), i le sommet de départ, j l'entier du caractère *)
}

(** Renvoie la liste de toutes les lignes dans le fichier [file_name]. *)
let read_file (file_name : string) : string list =
  let rec lire file liste =
    let line = input_line file in
    ();
    try lire file (line :: liste)
    with End_of_file ->
      close_in file;
      line :: liste
  in
```

```
  List.rev (lire (open_in file_name) [])

(** Renvoie la liste [[0...n-1]] *)
let range_list (n : int) : int list =
  let l = ref [] in
  for i = 0 to n - 1 do
    l := i :: !l
  done;
  !l

(** crée l'automate de stage 1 à partir d'une expression régulière [reg] et d'un
    terminal [t] *)
let automate_gen (reg : regex) (t : terminal) : automate =
  let dico = Hashtbl.create 0 in
  let a =
```

```
    ref { nodes = []; debut_l = [ 0 ]; fin = [ (1, t) ]; transitions = [||] }
  in

  (** ajoute la transition [n1 -> n2] étiquetée par [c] à l'automate *)
  let add_transition ((n1, c, n2) : int * char option * int) =
    if Hashtbl.mem dico n1 then
      Hashtbl.replace dico n1 ((c, n2) :: Hashtbl.find dico n1)
    else Hashtbl.add dico n1 [ (c, n2) ]
  in

  let next_node = ref 2 in

```

```
let rec automate_gen_aux (reg : regex) ((node_before, node_after) : int * int)
    : unit =
  match reg with
  | Vide -> ()
  | Epsilon -> add_transition (node_before, None, node_after)
  | AllChars ->
      for i = 32 to 127 do
        add_transition (node_before, Some (char_of_int i), node_after)
      done
  | Caractere x -> add_transition (node_before, Some x, node_after)
  | Concat (g, d) ->
      let node = !next_node in
      next_node := !next_node + 1;
```

```
            automate_gen_aux g (node_before, node);
            automate_gen_aux d (node, node_after)
        | Ou (g, d) ->
            automate_gen_aux g (node_before, node_after);
            automate_gen_aux d (node_before, node_after)
        | Range (a1, a2) ->
            for i = int_of_char a1 to int_of_char a2 do
              automate_gen_aux (Caractere (char_of_int i)) (node_before, node_after)
            done
        | ZeroPlus e ->
            automate_gen_aux e (node_before, node_before);
            automate_gen_aux Epsilon (node_before, node_after)
```

```
      | UnPlus e ->
        automate_gen_aux e (node_before,
         ↪  node_before);
        automate_gen_aux e (node_before,
         ↪  node_after)
      | Facultatif e ->
        automate_gen_aux e (node_before,
         ↪  node_after);
        automate_gen_aux Epsilon (node_before,
         ↪  node_after)
      | AllBut e ->
        for i = 0 to 127 do
          if e.(i) then
            add_transition (node_before, Some
             ↪  (char_of_int i), node_after)
        done
   in
```

```
automate_gen_aux reg (0, 1);

let l1 = List.of_seq (Hashtbl.to_seq_keys dico) in
let rec build_trans (l : int list) (arr : (char option * int) list array) :
    unit =
  match l with
  | [] -> ()
  | x :: q ->
      arr.(x) <- Hashtbl.find dico x;
      build_trans q arr
in
let arr = Array.make !next_node [] in
build_trans l1 arr;
{
  nodes = range_list !next_node;
```

```
    debut_l = !a.debut_l;
    fin = !a.fin;
    transitions = arr;
  }

(** construit la disjonction des automates de [l_a], qui reconnait donc l'union
    des language des automates de [l_a] *)
let ou_automates (l_a : automate list) : automate =
  (** construit la disjonction de la liste [l] avec un automate [out] *)
  let rec ou_automate_aux (l : automate list) (out : automate) =
    match l with
    | [] -> out
    | x :: q ->
        let inc = List.length out.nodes in
```

```
          let x2 =
            {
              nodes = List.map (( + ) inc) x.nodes;
              debut_l = List.map (( + ) inc)
               ↪  x.debut_l;
              fin = List.map (fun (x, y) -> (x + inc,
               ↪  y)) x.fin;
              transitions = [||];
            }
          in
          ou_automate_aux q
            {
              nodes = out.nodes @ x2.nodes;
              debut_l = out.debut_l @ x2.debut_l;
              fin = out.fin @ x2.fin;
              transitions =
                Array.init
```

```
                    (inc + Array.length x.transitions)
                    (fun i ->
                      if i < inc then out.transitions.(i)
                      else
                        List.map
                          (fun (c, x) -> (c, x + inc))
                          x.transitions.(i - inc));
              }
  in
  let a =
    ou_automate_aux l_a
      { nodes = []; debut_l = []; fin = []; transitions = [||] }
  in
  let nouv_d = List.length a.nodes in

```

```
(** Ajoute le début unique à l'automate par une transition [k -> x] étiquetée
    par epsilon (où k est l'état de départ, x est un ancien départ d'un
    élément de [l]) à out *)
let rec ajouter_debut (l : int list) (out : (char option * int) list array) :
    (char option * int) list array =
  Array.append out [| List.map (fun x -> (None, x)) l |]
in
{
  nodes = nouv_d :: a.nodes;
  debut_l = [ nouv_d ];
  fin = a.fin;
  transitions = ajouter_debut a.debut_l a.transitions;
```

```
  }

(** enlève les doublons dans la liste [l] *)
let remove_duplicates (l : 'a list) : 'a list =
  let tbl = Hashtbl.create 0 in
  let rec aux (l : 'a list) (out : 'a list) : 'a
   ↪  list =
    match l with
    | [] -> out
    | x :: q ->
        if Hashtbl.mem tbl x then aux q out
        else (
          Hashtbl.add tbl x 0;
          aux q (x :: out))
  in
  aux l []

```

```
(*
  Étapes pour enlever les epsilon-transitions:
  - On prend un sommet
  - On regarde toutes les epsilon-transitions
   ↪ sortantes (pas les boucles)
  - On regarde tous les transitions entrantes
  - On ajoute des nouvelles transitons des
   ↪ entrantes vers les sortantes
  - On supprime l'epsilon-transition

  Requis:
  - transitions entrantes de chaque sommet =>
   ↪ précalculé
*)

(** enlève les epsilon-transitions dans l'automate
↪ [a] en effectuant les étapes
```

```
    ci-dessus *)
let enleve_epsilon_trans (a : automate) : automate_sans_eps =
  let len = List.length a.nodes in
  (* on stocke les degrés entrants et sortants de chaque sommet *)
  (* les éléments stockés ne sont pas linéarisés *)
  let entrants = Array.make len [] in

  (* degré entrant de chaque sommet *)
  let degres = Array.make len 0 in

  let trans_temp = ref a.transitions in
  let fin_temp = ref a.fin in
  let deg_traite = ref 0 in

```

```
  (** trouve le premier sommet de degré zéro *)
  let rec find_premier_deg_zero () : int =
    let deg = !deg_traite in
    if degres.(!deg_traite) <= 0 then incr deg_traite;
    while !deg_traite < len && degres.(!deg_traite) > 0 do
      incr deg_traite
    done;
    if deg == !deg_traite then (
      incr deg_traite;
      find_premier_deg_zero ())
    else if !deg_traite >= len then -1
    else !deg_traite
  in

  (* construit les listes d'entrants de chaque sommet *)
```

```
Array.iteri
  (fun e l ->
    List.iter
      (fun (c, s) ->
        entrants.(s) <- (e, c) :: entrants.(s);
        degres.(s) <- degres.(s) + 1)
      l)
  a.transitions;
(* ajouter un de degré entrant pour chaque entrée *)
List.iter (fun x -> degres.(x) <- degres.(x) + 1) a.debut_l;

(* applique l'algorithme de suppression des epsilon transitions sur chacun des sommets *)
List.iter
  (fun node_i ->
```

```
    (* récupération les transitions entrantes
     ↪  avec epsilon transitions qui ne sont pas
     ↪  des boucles + enlever du degré entrant
     ↪  pour chaque epsilon transition *)
    let res =
      remove_duplicates
        (List.fold_left
           (fun acc x ->
             match x with
             | _, Some x -> acc
             | node, None ->
                 degres.(node_i) <-
                  ↪  degres.(node_i) - 1;
                 if node == node_i then acc else
                  ↪  node :: acc)
           [] entrants.(node_i))
    in
```

```
    (* on enlève les epsilon entrants de chaque epsilon transition *)
    entrants.(node_i) <-
      List.filter (fun (_, c) -> c != None) entrants.(node_i);
    (* on enlève les sortants de chaque epsilon transition *)
    !trans_temp.(node_i) <-
      List.filter
        (fun (c, x) -> not (c == None && x == node_i))
        !trans_temp.(node_i);
    List.iter
      (fun x ->
        !trans_temp.(x) <-
          List.filter
```

```
                (fun (c, x1) -> not (c == None && x1
                ↪  == node_i))
                !trans_temp.(x))
          res;
        (* ajoute les transitions des entrants avec
        ↪  epsilon transitions vers les sortants et
        ↪  actualise les entrants/sortants/degré de
        ↪  deux sommets *)
        List.iter
          (fun x ->
            List.iter
              (fun (c, node) ->
                !trans_temp.(x) <- (c, node) ::
                ↪  !trans_temp.(x);
                entrants.(node) <- (x, c) ::
                ↪  entrants.(node);
                degres.(node) <- degres.(node) + 1)
```

```
                (List.filter
                   (fun (c, n) -> not (n == node_i && c
                   ↪  = None))
                   !trans_temp.(node_i)))
            res;

        (* récupération des états de fin du sommet de
        ↪  départ *)
        let fins =
          List.fold_left
            (fun acc (x, t) -> if x == node_i then t
            ↪  :: acc else acc)
            [] a.fin
        in
```

```
    (* applique l'ensemble des états de fin aux sommets qui avaient une epsilon transition *)
    List.iter
      (fun x -> List.iter (fun t -> fin_temp := (x, t) :: !fin_temp) fins)
      res)
  a.nodes;

(* enlever les nodes qui ne sont plus atteintes *)
let todo = ref (find_premier_deg_zero ()) in
while !todo <> -1 do
  List.iter (fun (_, x) -> degres.(x) <- degres.(x) - 1) !trans_temp.(!todo);
  todo := find_premier_deg_zero ()
done;
```

```

    (* enlever les option car toutes les transition (devraient être) sans epsilon transition *)
    {
      nodes = List.filter (fun x -> degres.(x) > 0) a.nodes;
      debut_l = a.debut_l;
      fin = List.filter (fun (x, _) -> degres.(x) > 0) !fin_temp;
      transitions_sans_eps =
        Array.map
          (fun x ->
            List.map
              (fun (c, n) ->
                match c with None -> failwith "pas correct" | Some c1 -> (c1, n))
```

```
             (List.filter (fun (_, n) -> degres.(n) > 0) x))
         !trans_temp;
  }

(** Donne à chaque ensemble d'éléments [elem] un entier à partir de la table de
    linéarisation [lin_tbl] et cette de délinéarisation [delin_tbl] *)
let lin (elem : IntSet.t) (lin_tbl : (IntSet.t, int) Hashtbl.t)
    (delin_tbl : IntSet.t Vector.t) : int =
  if not (Hashtbl.mem lin_tbl IntSet.empty) then
    Hashtbl.add lin_tbl IntSet.empty 0;

  if Hashtbl.find lin_tbl IntSet.empty <> Vector.length delin_tbl then
```

```
    failwith "La taille des deux tables n'est pas la même";

  if not (Hashtbl.mem lin_tbl elem) then (
    Hashtbl.add lin_tbl elem (Hashtbl.find lin_tbl IntSet.empty);
    Hashtbl.replace lin_tbl IntSet.empty (Hashtbl.find lin_tbl IntSet.empty + 1);
    Vector.push delin_tbl elem);
  Hashtbl.find lin_tbl elem

(** Pour chaque entier [elem], renvoie l'ensemble de sommets corrspondant, à
    l'aide de la table de délinéarisation [delin_tbl] *)
let delin (elem : int) (delin_tbl : IntSet.t Vector.t) : IntSet.t =
```

```
    if elem = -1 then IntSet.empty
    else if elem >= Vector.length delin_tbl then
      failwith "L'élément demandé n'est pas dans la table"
    else Vector.get delin_tbl elem

(** déterminise l'automate [a] *)
let determinise (a : automate_sans_eps) : automate_det =
  (* crée les deux table utiles pour la linéarisation / délinéarisation *)
  let lin_tbl = Hashtbl.create (List.length a.nodes) in
  let delin_tbl = Vector.create ~dummy:IntSet.empty in

```

```
  let start_node = lin (IntSet.of_list a.debut_l)
   ↪  lin_tbl delin_tbl in
  let todo = ref [ start_node ] in

  let a_det =
    {
      nodes = [ start_node ];
      debut = start_node;
      fin = [||];
      pre_transitions = Vector.create
       ↪  ~dummy:(Array.make 1 0);
    }
  in

  Vector.push a_det.pre_transitions (Array.make 128
   ↪  (-1));
  (* pour le début *)
```

```
  let fin = IntSet.of_list (List.map (fun (a, b) -> a) a.fin) in

  (** trouve les successeurs de tous les sommets de [l] et on les stocke dans
      [arr] *)
  let rec trouver_suivants (l : IntSet.t) (arr : int array) : unit =
    let len = Array.length arr in

    (* on rassemble les éléments accessibles depuis tous les sommets de l *)
    let storage = Array.make len IntSet.empty in
    IntSet.iter
      (fun x ->
        List.iter
          (fun (c, e) ->
```

```
              storage.(int_of_char c) <- IntSet.add e storage.(int_of_char c))
            a.transitions_sans_eps.(x))
        l;

      (* on linéarise les sommets obtenus et on les stocke dans arr *)
      for i = 0 to len - 1 do
        if not (IntSet.is_empty storage.(i)) then
          arr.(i) <- lin storage.(i) lin_tbl delin_tbl
      done
    in

    (** teste si le sommet [elem] linéarisé contient des éléments finaux et
        l'ajoute aux finaux si c'est le cas *)
```

```
  let ajouter_fin (elem : int) : unit =
    let res =
      remove_duplicates
        (List.map
           (fun e -> List.assoc e a.fin)
           (IntSet.to_list (IntSet.inter fin (delin elem delin_tbl))))
    in
    match res with
    | [] -> ()
    | [ e ] -> a_det.fin.(elem) <- Some e
    | [ e1; e2 ] ->
        (* L'un des deux est un safe_token, elle peut être vue autrement donc elle est ignorée *)
        if e1 = safe_token then a_det.fin.(elem) <- Some e2
```

```
        else if e2 = safe_token then a_det.fin.(elem) <- Some e1
        else (
          print_char '"';
          print_string (repr_of_terminal e1);
          print_string "\" \"";
          print_string (repr_of_terminal e2);
          print_char '"';
          print_string "on un état final commun";
          print_newline ();
          failwith "" (* il a plus d'un élément final *))
    | _ -> failwith "A syntax can't have more than one output"
  in

  let finished = ref false in
```

```
    let seen = ref IntSet.empty in
    (* on construit l'automate déterministe *)
    while not !finished do
      match !todo with
      | [] -> finished := true
      | x :: q ->
            let init_len = Hashtbl.find lin_tbl IntSet.empty in
            todo := q;
            let suivants = Array.make 128 (-1) in
            trouver_suivants (delin x delin_tbl) suivants;

            let arr = Vector.get a_det.pre_transitions x in
            for i = 0 to 127 do
              if suivants.(i) >= init_len then
```

```
          (* si c'est un nouveau noeud, on l'ajoute a la liste de traitement, on l'ajoute dans l'automate et on vérifie s'il est final *)
          if not (IntSet.mem suivants.(i) !seen) then (
            seen := IntSet.add suivants.(i) !seen;
            todo := suivants.(i) :: !todo;
            Vector.push a_det.pre_transitions (Array.make 128 (-1));
            a_det.nodes <- suivants.(i) :: a_det.nodes);
        (* noeud déjà existant/complétion nouveau noeud, comme on ne traite qu'une fois chaque sommets,
```

```
            on sait que les sommets trouvé sont les bons, on les remplace *)
            if suivants.(i) <> -1 then arr.(i) <- suivants.(i)
          done;
          Vector.set a_det.pre_transitions x arr
  done;

  (* gérer le cas des mots vides, qui sont donc à la fois initiaux et finaux *)
  a_det.fin <- Array.make (List.length a_det.nodes) None;
  List.iter ajouter_fin a_det.nodes;
  {
    nodes = a_det.nodes;
    debut = a_det.debut;
    fin = a_det.fin;
```

```
    transitions = Vector.to_array a_det.pre_transitions;
  }

(** effectue le delta 1 sur l'automate [a] à partir de [node] avec l'étiquette
    [c] *)
let exec_char (a : automate_det) (node : int) (c : char) : int =
  a.transitions.(node).(int_of_char c)

(** effectue le delta étoile sur l'automate [a] avec le texte [texte] *)
let execution_mot (a : automate_det) (texte : char list) :
    int * char list * char list =
  let node = ref a.debut in
```

```
  let last_found = ref (-1) in
  let text_as_last = ref texte in
  let texte = ref texte in
  let text_read = ref [] in
  let last_read = ref [] in

  (* tant que l'on est pas dans le puit *)
  while !node != -1 do
    match !texte with
    | [] -> node := -1 (* forcer la fin de la boucle *)
    | c :: q ->
        (* on effectue le delta 1 *)
        text_read := c :: !text_read;
        node := exec_char a !node c;
        texte := q;
        (* on note si on passe par un état final *)
```

```
            if !node = -1 then (if !last_found = -1 then last_read := !text_read)
            else (
              (match a.fin.(!node) with
              | None -> ()
              | Some t ->
                  last_found := !node;
                  text_as_last := !texte);
              last_read := !text_read)
  done;
  (* on renvoie le dernier état final trouvé*)
  (!last_found, !text_as_last, !last_read)

(** exécute l'automate [a] en boucle sur le texte [txt] pour créer une liste de
    lexèmes *)
let exec (a : automate_det) (txt : string) : (symbol * string) list =
```

```
(** exécute l'automate [a] en boucle sur le texte [texte] et concatène le
    résultat dans [out] *)
let rec exec_aux (a : automate_det) (texte : char list)
    (out : (terminal * string) list) : (terminal * string) list =
  match texte with
  | [] -> List.rev out
  | _ -> (
      match execution_mot a texte with
      | -1, _, s ->
          print_string "Le lexème '";
          print_string (String.of_seq (List.to_seq (List.rev s)));
          print_string "' n'est pas un lexème reconnu\n";
```

```
                failwith ""
            | x, q, s -> (
                let s = String.of_seq (List.to_seq
                 ↪  (List.rev s)) in
                match a.fin.(x) with
                | None ->
                    print_string "Le lexème '";
                    print_string s;
                    print_string "' n'est pas un lexème
                     ↪  reconnu\n";
                    failwith ""
                | Some t -> exec_aux a q ((t, s) ::
                 ↪  out)))
    in
    let res = exec_aux a (List.of_seq (String.to_seq
     ↪  txt)) [] in
```

```
  let tbl = Hashtbl.create (List.length unparsed_tokens) in
  List.iter (fun x -> Hashtbl.add tbl x ()) unparsed_tokens;
  List.map
    (fun (t, s) -> (Terminal t, s))
    (List.filter (fun (x, _) -> not (Hashtbl.mem tbl x)) res)

(** exécute l'automate [a] sur le fichier [f_name] *)
let exec_of_file (a : automate_det) (f_name : string) : (symbol * string) list =
  exec a
    (List.fold_left (fun acc line -> acc ^ line ^ "\n") "" (read_file f_name))
```

```ocaml
open Grammar_functions
open Symbols

module SymbolSet = Set.Make (struct
  type t = symbol

  let compare = compare
end)

(* arbre de syntaxe (non abstraite) *)
type at = Noeud of (symbol * string) * at list
type symbol_SS_Htbl = (symbol, SymbolSet.t)
↪  Hashtbl.t

let print_SymbolSet (ss : SymbolSet.t) : unit =
  print_endline "SymbolSet(";
  SymbolSet.iter
```

```
    (fun s ->
      print_symbol s;
      print_newline ())
    ss;
  print_endline ")\n"

let print_SymbolSet_Hastable (h : symbol_SS_Htbl) :
↪  unit =
  Hashtbl.iter
    (fun s ss ->
      print_string (string_of_symbol s ^ " -> ");
      print_SymbolSet ss;
      print_newline ())
    h

(* returns a hashtable containing the first set of
↪  every non terminal *)
```

```
let first (g : grammar) : symbol_SS_Htbl =
  (* prog dyn, filling first_h *)
  let first_h = Hashtbl.create (Hashtbl.length g.rules_htbl) in

  let rec first_of_rule ((s, patterns) : rule) : unit =
    if is_terminal (s, patterns) then
      Hashtbl.add first_h s (SymbolSet.singleton s)
    else if
      (* checking if already computed *)
      not (Hashtbl.mem first_h s)
    then
      Hashtbl.add first_h s
        (List.fold_left
          (fun acc d ->
            let f = first_of_pattern s d in
```

```
                    if SymbolSet.disjoint f acc then SymbolSet.union f acc
                    else (
                      print_symbol s;
                      print_newline ();
                      print_SymbolSet f;
                      prerr_newline ();
                      print_SymbolSet acc;
                      print_newline ();
                      failwith "the first set is not disjoined"))
                SymbolSet.empty patterns)
    else ()
  and first_of_pattern (s : symbol) (d : pattern) : SymbolSet.t =
    match d with
    | [] -> failwith "empty pattern"
```

```
    | Terminal t :: q -> SymbolSet.singleton
    ↪  (Terminal t)
    | fst_symbol :: q ->
      first_of_rule (fst_symbol, Hashtbl.find
      ↪  g.rules_htbl fst_symbol);
      if
        SymbolSet.mem (Terminal E) (Hashtbl.find
        ↪  first_h fst_symbol)
        && q <> []
      then
        SymbolSet.union (first_of_pattern s q)
          (SymbolSet.remove (Terminal E)
          ↪  (Hashtbl.find first_h fst_symbol))
      else Hashtbl.find first_h fst_symbol
  in
  Hashtbl.iter (fun name pattern -> first_of_rule
  ↪  (name, pattern)) g.rules_htbl;
```

```
  first_h

let rec first_of_pattern (fst_ss_htbl : symbol_SS_Htbl) (p : pattern) : SymbolSet.t =
  (* print_endline ("first_of_pattern "); print_patterns [p];print_newline (); *)
  match p with
  | [] -> failwith "empty pattern in first_of_pattern"
  | Terminal t :: _ -> SymbolSet.singleton (Terminal t)
  | NonTerminal nt :: q ->
     if
       SymbolSet.mem (Terminal E) (Hashtbl.find fst_ss_htbl (NonTerminal nt))
       && q <> []
```

```
        then
          SymbolSet.union
            (first_of_pattern fst_ss_htbl q)
            (SymbolSet.remove (Terminal E)
               (Hashtbl.find fst_ss_htbl (NonTerminal
               ↪  nt)))
        else Hashtbl.find fst_ss_htbl (NonTerminal
        ↪  nt)

(* returns a hashtable containing the follow set of
↪  every non terminal (all the terminals that can
↪  occur after a rule) *)
let follow (g : grammar) : symbol_SS_Htbl =
  (* contains the terminals following the key in
  ↪  the hashtable *)
  let follow_non_terminal =
    Hashtbl.of_seq
```

```
        (Seq.map
           (fun (name, pattern) -> (name, SymbolSet.empty))
           (Hashtbl.to_seq g.rules_htbl))
    in

    (* contains the rule names in which the key appears in at the end *)
    let parent_on_end =
      Hashtbl.of_seq
        (Seq.map
           (fun (name, pattern) -> (name, SymbolSet.empty))
           (Hashtbl.to_seq g.rules_htbl))
    in

```

```
(* fills the follow_non_terminal and parent_on_end *)
let rec pattern_follow (s : symbol) (d : pattern) : unit =
  match d with
  | NonTerminal s1 :: s2 :: q ->
      Hashtbl.replace follow_non_terminal (NonTerminal s1)
        (SymbolSet.add s2 (Hashtbl.find follow_non_terminal (NonTerminal s1)));
      pattern_follow s (s2 :: q)
  | Terminal s1 :: q -> pattern_follow s q
  | NonTerminal s1 :: [] ->
      if NonTerminal s1 <> s && NonTerminal s1 <> Terminal E then
```

```
            Hashtbl.replace parent_on_end
             ↪  (NonTerminal s1)
              (SymbolSet.add s (Hashtbl.find
               ↪  parent_on_end (NonTerminal s1)))
          else ()
      | [] -> ()
  in

  let patterns_follow (s : symbol) (patterns :
   ↪  pattern list) : unit =
    List.iter (pattern_follow s) patterns
  in
  Hashtbl.iter patterns_follow g.rules_htbl;

  let first_h = first g in
  (* contains the terminals that can appear after a
   ↪  key in the grammar*)
```

```
  let follow_h = Hashtbl.create (Hashtbl.length first_h) in

  let rec union_and_next_on_epsilon (next_hashtable : symbol_SS_Htbl)
      (s : symbol) (acc : SymbolSet.t) : SymbolSet.t =
    if is_non_terminal_symbol s then
      if SymbolSet.mem (Terminal E) (Hashtbl.find next_hashtable s) then (
        follow_of_symbol s;
        SymbolSet.remove (Terminal E)
          (SymbolSet.union acc
             (SymbolSet.union (Hashtbl.find follow_h s)
                (Hashtbl.find next_hashtable s))))
```

```
        else SymbolSet.union acc (Hashtbl.find
         ↪  next_hashtable s)
      else SymbolSet.singleton s
    and follow_of_symbol (s : symbol) : unit =
      (* checking if already computed *)
      if not (Hashtbl.mem follow_h s) then
        let follow_non_terminal_set =
          SymbolSet.fold
            (union_and_next_on_epsilon first_h)
            (Hashtbl.find follow_non_terminal s)
            (SymbolSet.singleton (Terminal EOF))
        in
        let follow_parent_on_end_set =
          SymbolSet.fold
            (fun s ->
              follow_of_symbol s;
              union_and_next_on_epsilon follow_h s)
```

```
          (Hashtbl.find parent_on_end s)
          (SymbolSet.singleton (Terminal EOF))
      in

      Hashtbl.replace follow_h s
        (SymbolSet.union follow_non_terminal_set follow_parent_on_end_set)
    else ()
  in
  Hashtbl.iter (fun s _ -> follow_of_symbol s) g.rules_htbl;
  follow_h

let analyse_LL1_of_symbol (g : grammar) (text : (symbol * string) list)
    (s : symbol) : at * (symbol * string) list =
  let follow_sshtbl = follow g in
```

```
  let first_sshtbl = first g in

  let rec analyse_LL1_of_pattern (text : (symbol *
   ↪  string) list) (s : symbol)
      (p : pattern) : at * (symbol * string) list =
    (* print_endline "analyse_LL1_of_pattern"; *)
    if p = [ Terminal E ] then
      (Noeud ((s, ""), [ Noeud ((Terminal E, ""),
       ↪  []) ]), text)
    else
      let txt = ref text in
      let t =
        Noeud
          ( (s, ""),
            List.map
              (fun (s_p : symbol) ->
```

```
                let tree, txt' = analyse_LL1_of_symbol_aux !txt s_p in
                txt := txt';
                tree)
              p )
    in
    (t, !txt)
and analyse_LL1_of_symbol_aux (text : (symbol * string) list) (s : symbol) :
    at * (symbol * string) list =
   (* print_endline ("analyse_LL1_of_symbol_aux " ^ string_of_symbol s); *)
   (*print_patterns [ List.map fst text ];
   print_newline ();*)
   if is_terminal (s, []) then
     (* print_endline "is terminal"; *)
```

```
      match text with
      | [] -> failwith "no text to match in analyse_LL1, text is empty"
      | t1 :: q ->
          if fst t1 = s then (Noeud (t1, []), q)
          else failwith "the expected terminal does not match the text"
    else
      (* print_endline "is non_terminal"; *)
      match text with
      | [] ->
          if
            (* print_endline "Terminal E"; *)
            SymbolSet.mem (Terminal E) (Hashtbl.find first_sshtbl s)
            && SymbolSet.mem (Terminal EOF) (Hashtbl.find follow_sshtbl s)
```

```
        then (
          let l =
            List.filter
              (fun p ->
                SymbolSet.mem (Terminal E) (first_of_pattern first_sshtbl p))
              (Hashtbl.find g.rules_htbl s)
          in
          if l = [] then (
            print_string
              ("on " ^ string_of_symbol s ^ ", expected "
              ^ string_of_symbol (Terminal E)
              ^ " in first of patterns but first of patters is\n");
```

```
              print_SymbolSet (Hashtbl.find first_sshtbl s);
              failwith "error 1")
            else ();
            analyse_LL1_of_pattern text s (List.nth l 0))
          else failwith "text is epsilon but more symbols are expected"
      | _ ->
          if
            (*print_SymbolSet (Hashtbl.find first_sshtbl s);
            print_SymbolSet (Hashtbl.find follow_sshtbl s);*)
            SymbolSet.mem (Terminal E) (Hashtbl.find first_sshtbl s)
            && SymbolSet.mem
```

```
                    (fst (List.nth text 0))
                    (Hashtbl.find follow_sshtbl s)
             then (
               let l =
                 List.filter
                   (fun p ->
                     SymbolSet.mem (Terminal E) (first_of_pattern first_sshtbl p))
                   (Hashtbl.find g.rules_htbl s)
               in
               if l = [] then (
                 print_string
                   ("on " ^ string_of_symbol s ^ ", expected "
                   ^ string_of_symbol (Terminal E)
```

```
              ^ " in first of patterns but first of patters is\n");
            print_SymbolSet (Hashtbl.find first_sshtbl s);
            failwith "error 2")
          else ();
          analyse_LL1_of_pattern text s (List.nth l 0))
        else (
          (* print_endline "cas 2";
          print_symbol (fst (List.nth text 0));
          print_newline (); *)
          let l =
            List.filter
              (fun p ->
                SymbolSet.mem
                  (fst (List.nth text 0))
```

```
                (first_of_pattern first_sshtbl
                 ↪  p))
              (Hashtbl.find g.rules_htbl s)
          in
          if l = [] then (
            print_string
              ("on " ^ string_of_symbol s ^ ",
               ↪  expected "
              ^ string_of_symbol (fst (List.nth
               ↪  text 0))
              ^ " in first of patterns but first
               ↪  of patters is\n");
            print_SymbolSet (Hashtbl.find
             ↪  first_sshtbl s);
            failwith "error 3")
          else ();
```

```
            analyse_LL1_of_pattern text s (List.nth l 0))
  in
  analyse_LL1_of_symbol_aux text s

let analyse_LL1 (g : grammar) (text : (symbol * string) list) : at =
  fst (analyse_LL1_of_symbol g (text @ [ (Terminal EOF, "") ]) g.start_symbol)
```

```ocaml
open Abstract_tokens
open LL1
open Symbols

(** écrase les noeuds à écraser à partir de [t] et l'ajoute à ceux de la liste
    [l] (pour remonter des arguments par exemple) *)
let flatten (t : ast) (l : ast list) : ast list =
  let rec aux (t : ast) (out : ast list) : ast list =
    match t with
    | Noeud (ToFlatten, []) -> out
    | Noeud (ToFlatten, x :: q) ->
        (* vide l'entièreté des éléments de x dans out puis ceux de q *)
        let out = aux x out in
```

```
        aux (Noeud (ToFlatten, q)) out
      | _ -> t :: out
  in
  List.rev_append (aux t []) l

(** convertis l'arbre de syntaxe abstrait [t] pour
↪  que les fonctions, dont le
    nom est stocké au fur et à mesure dans
      ↪  [curr_func], aient leur valeur de
    retour *)
let rec link_return_function (t : ast) (return_val
↪  : string option) : ast =
  match t with
  | Noeud (Syntax Function, l) ->
      let name, l =
        List.fold_left
          (fun (_name, _l) (x : ast) ->
```

```
          match x with
          | Noeud (Name s, []) when _name = "" -> (s, x :: _l)
          | Noeud (Syntax Out, [ Noeud (Name s, []) ]) -> (s, _l)
          | _ -> (_name, x :: _l))
        ("", []) l
    in
    let l =
      match l with
      | Noeud (Syntax Return, []) :: q -> List.rev l
      | _ ->
          let (l : ast list) =
            Noeud (Syntax Return, [ Noeud (Name name, []) ]) :: l
          in
```

```
         List.rev l
     in
     Noeud
       ( Syntax Function,
        List.map (fun x -> link_return_function x (Some name)) l )
  | Noeud (Syntax Subroutine, l) ->
     Noeud
       ( Syntax Function,
        match List.rev l with
        | Noeud (Syntax Return, []) :: q -> List.rev q
        | _ -> l )
  | Noeud (Syntax Return, []) -> (
     match return_val with
     | None -> failwith "Return doit être dans une fonction"
```

```
        | Some s -> Noeud (Syntax Return, [ Noeud
        ↪  (Name s, []) ]))
    | Noeud (x, l) ->
        Noeud (x, List.map (fun x ->
        ↪  link_return_function x return_val) l)

(** Si [t] est un commentaire, le convertit pour
↪  séparer les différentes lignes,
    sinon renvoie une erreur *)
let convert_comments (t : ast) : ast list =
  let nb = ref 0 in
  match t with
    | Noeud (Commentaire s, []) ->
        let l = List.map String.trim
        ↪  (String.split_on_char '\n' s) in
        let l2 =
          List.map
```

```
          (fun (s : string) : ast ->
            if s = "" then Noeud (NewLine, [])
            else (
              incr nb;
              Noeud (Commentaire (String.sub s 1 (String.length s - 1)), []))
          l
      in
      if !nb = 0 then match l2 with [] -> l2 | _ :: q -> q else l2
  | _ -> failwith "l'argument donné n'est pas un commentaire"

(** Enlève le niveau d'abstraction de l'arbre de syntaxe [t] créé avec LL1 *)
let convert_to_abstract (t : at) : ast =
  let rec convert_to_abstract_aux (t : at) : ast =
```

```
    match t with
    | Noeud ((NonTerminal ExecutableProgram, _), l) -> (
        match l with
        | [
          Noeud ((NonTerminal StartCommentBlock, s1), l1);
          Noeud ((NonTerminal Function_or_Subroutine_star_MainProgram, s), l);
        ] ->
            Noeud
              ( ProgramRoot,
                convert_to_abstract_aux (Noeud ((Terminal EOS, s1), []))
                :: flatten
                     (convert_to_abstract_aux
```

```
                                    (Noeud
                                        ( ( NonTerminal
                                              
                                                ↪  Function_or_Subrout
                                                s ),
                                            l )))
                                []  )
                    | [
                    Noeud ((NonTerminal
                     ↪  Function_or_Subroutine_star_MainProgram,
                     ↪  s), l);
                    ] ->
                        Noeud
                           ( ProgramRoot,
                             flatten
                                (convert_to_abstract_aux
                                    (Noeud
```

```
                              ( ( NonTerminal Function_or_Subroutine_star_
                                  s ),
                                l )))
                        [] )
            | _ -> failwith "ExecutableProgram")
        | Noeud ((NonTerminal Function_or_Subroutine_star_MainProgram, _), l) -> (
            match l with
            | [
             Noeud ((NonTerminal Recursive_opt_Function_or_Subroutine, s), l);
             Noeud ((NonTerminal Function_or_Subroutine_star_MainProgram, s1), l1);
```

```
    ] ->
        Noeud
          ( ToFlatten,
            [
              convert_to_abstract_aux
                (Noeud
                   ((NonTerminal
                   ↪   Recursive_opt_Function_or_Sul
                   ↪   s), l));
              convert_to_abstract_aux
                (Noeud
                   ( ( NonTerminal
                   ↪   Function_or_Subroutine_star_I
                       s1 ),
                     l1 ));
            ] )
    | [
```

```
        Noeud ((NonTerminal MainProgram, s), l);
        Noeud
          ( (NonTerminal Function_or_Subroutine_star, _),
            [ Noeud ((Terminal E, _), []) ] );
      ] ->
          convert_to_abstract_aux (Noeud ((NonTerminal MainProgram, s), l))
      | [
        Noeud ((NonTerminal MainProgram, s), l);
        Noeud ((NonTerminal Function_or_Subroutine_star, s1), l1);
      ] ->
          Noeud
            ( ToFlatten,
              [
                convert_to_abstract_aux
```

```
                    (Noeud ((NonTerminal MainProgram, s), l));
                  convert_to_abstract_aux
                    (Noeud ((NonTerminal Function_or_Subroutine_star, s1), l1));
              ] )
          | _ -> failwith "Function_or_Subroutine_star_MainProgram")
      | Noeud ((NonTerminal Function_or_Subroutine_star, _), l) -> (
        match l with
          | [
            Noeud ((NonTerminal Recursive_opt_Function_or_Subroutine, s), l);
            Noeud
```

```
            ( (NonTerminal Function_or_Subroutine_star, _),
              [ Noeud ((Terminal E, _), []) ] );
        ] ->
            convert_to_abstract_aux
              (Noeud ((NonTerminal Recursive_opt_Function_or_Subroutine, s), l))
        | [
          Noeud ((NonTerminal Recursive_opt_Function_or_Subroutine, s), l);
          Noeud ((NonTerminal Function_or_Subroutine_star, s1), l1);
        ] ->
            Noeud
              ( ToFlatten,
```

```
                        [
                          convert_to_abstract_aux
                            (Noeud
                               ((NonTerminal
                               ↪  Recursive_opt_Function_or_Su
                               ↪  s), l));
                          convert_to_abstract_aux
                            (Noeud ((NonTerminal
                            ↪  Function_or_Subroutine_star,
                            ↪  s1), l1));
                        ] )
                | _ -> failwith
                ↪  "Function_or_Subroutine_star")
            | Noeud
                ( (NonTerminal
                ↪  Recursive_opt_Function_or_Subroutine,
                ↪  _),
```

```
            [
              Noeud ((Terminal Recursive, _), []);
              Noeud ((NonTerminal Function_or_Subroutine, s), l);
            ] )
      | Noeud
          ( (NonTerminal Recursive_opt_Function_or_Subroutine, _),
            [ Noeud ((NonTerminal Function_or_Subroutine, s), l) ] ) ->
          convert_to_abstract_aux
            (Noeud ((NonTerminal Function_or_Subroutine, s), l))
      | Noeud
          ( (NonTerminal Function_or_Subroutine, _),
```

```
              [ Noeud ((NonTerminal FunctionSubprogram,
              ↪  s), l) ] ) ->
            convert_to_abstract_aux (Noeud
            ↪  ((NonTerminal FunctionSubprogram, s),
            ↪  l))
        | Noeud
            ( (NonTerminal Function_or_Subroutine, _),
              [ Noeud ((NonTerminal
              ↪  SubroutineSubprogram, s), l) ] ) ->
            convert_to_abstract_aux
              (Noeud ((NonTerminal
              ↪  SubroutineSubprogram, s), l))
        | Noeud
            ( (NonTerminal FunctionSubprogram, _),
              [
                Noeud ((NonTerminal FunctionPrefix, s),
                ↪  l);
```

```
                Noeud ((NonTerminal FunctionName, s1),
                 ↪  l1);
                Noeud ((NonTerminal FunctionRange, s2),
                 ↪  l2);
              ] ) -> (
          match l with
          | [ Noeud ((Terminal Function, _), []) ] ->
              Noeud
                ( Syntax Function,
                  flatten
                    (Noeud
                       ( ToFlatten,
                         [
                           convert_to_abstract_aux
                             (Noeud ((NonTerminal
                              ↪  FunctionName, s1),
                              ↪  l1));
```

```
                                        convert_to_abstract_aux
                                          (Noeud ((NonTerminal FunctionRange, s2), l2));
                                      ] ))
                                 [] )
                    | _ ->
                        Noeud
                          ( Syntax Function,
                            flatten
                              (Noeud
                                 ( ToFlatten,
                                   [
                                     convert_to_abstract_aux
                                       (Noeud ((NonTerminal FunctionPrefix, s), l));
```

```
                                   convert_to_abstract_aux
                                     (Noeud ((NonTerminal
                                     ↪   FunctionName, s1),
                                     ↪   l1));
                                   convert_to_abstract_aux
                                     (Noeud ((NonTerminal
                                     ↪   FunctionRange, s2),
                                     ↪   l2));
                                 ] ))
                             [] ))
      | Noeud
          ( (NonTerminal FunctionPrefix, _),
            [
              Noeud ((NonTerminal TypeSpec, s), l);
              Noeud ((Terminal Function, _), []);
            ] ) ->
```

```
        convert_to_abstract_aux (Noeud ((NonTerminal TypeSpec, s), l))
    | Noeud
        ( (NonTerminal FunctionRange, _),
          [
            Noeud ((NonTerminal FunctionParList, s), l);
            Noeud ((NonTerminal FunctionResult_opt, s4), l4);
            Noeud ((Terminal EOS, s1), l1);
            Noeud ((NonTerminal BodyConstruct_star, s3), l3);
            Noeud ((NonTerminal EndFunctionStmt, s2), l2);
          ] ) ->
        let n = ref [] in
        let n3 = ref [] in
```

```
            let n4 = ref [] in
            (match l3 with
            | [ Noeud ((Terminal E, _), []) ] -> ()
            | _ ->
                n3 :=
                  [
                    convert_to_abstract_aux
                      (Noeud ((NonTerminal
                        ↪  BodyConstruct_star, s3),
                        ↪  l3));
                  ]);
            (match l with
            | [
             Noeud ((Terminal LParenthesis, _), []);
             Noeud
```

```
              ( (NonTerminal
               ↪  FunctionPar_Comma_FunctionPar_star_opt,
               ↪  _),
                [ Noeud ((Terminal E, _), []) ] );
            Noeud ((Terminal RParenthesis, _), []);
          ] ->
              ()
          | _ ->
              n :=
                [
                  convert_to_abstract_aux
                    (Noeud ((NonTerminal
                     ↪  FunctionParList, s), l));
                ]);
          (match l4 with
          | [ Noeud ((Terminal E, _), []) ] -> ()
          | _ ->
```

```
            n4 :=
              [
                convert_to_abstract_aux
                  (Noeud ((NonTerminal FunctionResult_opt, s4), l4));
              ]);
        Noeud
          ( ToFlatten,
            flatten
              (Noeud (ToFlatten, !n))
              (flatten
                 (Noeud (ToFlatten, !n4))
                 (convert_to_abstract_aux (Noeud ((Terminal EOS, s1), l1))
                 :: flatten
                      (Noeud (ToFlatten, !n3))
```

```
                      [
                        convert_to_abstract_aux
                          (Noeud ((NonTerminal EndFunctionStmt, s2), l2));
                      ])) )
    | Noeud
        ( (NonTerminal FunctionResult_opt, _),
          [
            Noeud ((Terminal Result, _), []);
            Noeud ((Terminal LParenthesis, _), []);
            Noeud ((NonTerminal VariableName, s), l);
            Noeud ((Terminal RParenthesis, _), []);
          ] ) ->
        Noeud
          ( Syntax Out,
```

```
            [
              convert_to_abstract_aux (Noeud
              ↪  ((NonTerminal VariableName, s),
              ↪  l));
            ] )
    | Noeud
        ( (NonTerminal FunctionParList, _),
          [
            Noeud ((Terminal LParenthesis, _), []);
            Noeud ((NonTerminal
            ↪  FunctionPar_Comma_FunctionPar_star_opt,
            ↪  s), l);
            Noeud ((Terminal RParenthesis, _), []);
          ] ) ->
        convert_to_abstract_aux
```

```
          (Noeud ((NonTerminal FunctionPar_Comma_FunctionPar_star_opt, s), l))
    | Noeud
        ( (NonTerminal FunctionPar_Comma_FunctionPar_star_opt, _),
          [
            Noeud ((NonTerminal FunctionPar, s), l);
            Noeud
              ( (NonTerminal Comma_FunctionPar_star, _),
                [ Noeud ((Terminal E, _), []) ] );
          ] ) ->
        convert_to_abstract_aux (Noeud ((NonTerminal FunctionPar, s), l))
```

```
      | Noeud
          ( (NonTerminal
           ↪  FunctionPar_Comma_FunctionPar_star_opt,
           ↪  _),
            [
              Noeud ((NonTerminal FunctionPar, s),
               ↪  l);
              Noeud ((NonTerminal
               ↪  Comma_FunctionPar_star, s1), l1);
            ] ) ->
          Noeud
            ( ToFlatten,
              [
                convert_to_abstract_aux (Noeud
                 ↪  ((NonTerminal FunctionPar, s),
                 ↪  l));
                convert_to_abstract_aux
```

```
                  (Noeud ((NonTerminal Comma_FunctionPar_star, s1), l1));
              ] )
        | Noeud
            ( (NonTerminal Comma_FunctionPar_star, _),
              [
                Noeud ((Terminal Comma, _), []);
                Noeud ((NonTerminal FunctionPar, s), l);
                Noeud
                  ( (NonTerminal Comma_FunctionPar_star, _),
                    [ Noeud ((Terminal E, _), []) ] );
              ] ) ->
            convert_to_abstract_aux (Noeud ((NonTerminal FunctionPar, s), l))
```

```
    | Noeud
        ( (NonTerminal Comma_FunctionPar_star, _),
          [
            Noeud ((Terminal Comma, _), []);
            Noeud ((NonTerminal FunctionPar, s), l);
            Noeud ((NonTerminal Comma_FunctionPar_star, s1), l1);
          ] ) ->
        Noeud
          ( ToFlatten,
            [
              convert_to_abstract_aux (Noeud ((NonTerminal FunctionPar, s), l));
              convert_to_abstract_aux
```

```
                    (Noeud ((NonTerminal Comma_FunctionPar_star, s1), l1));
                ] )
        | Noeud
            ( (NonTerminal FunctionPar, _),
              [ Noeud ((NonTerminal DummyArgName, s), l) ] ) ->
            convert_to_abstract_aux (Noeud ((NonTerminal DummyArgName, s), l))
        | Noeud
            ( (NonTerminal EndFunctionStmt, _),
              [
                Noeud ((Terminal EndFunction, _), []);
                Noeud ((NonTerminal EndName_opt, _), _);
                Noeud ((Terminal EOS, s), l);
```

```
          ] ) ->
        convert_to_abstract_aux (Noeud ((Terminal EOS, s), l))
    | Noeud
        ( (NonTerminal SubroutineSubprogram, _),
          [
            Noeud ((Terminal Subroutine, _), []);
            Noeud ((NonTerminal SubroutineName, s), l);
            Noeud ((NonTerminal SubroutineRange, s1), l1);
          ] ) ->
        Noeud
          ( Syntax Subroutine,
            convert_to_abstract_aux (Noeud ((NonTerminal SubroutineName, s), l))
```

```
                    :: flatten
                         (convert_to_abstract_aux
                            (Noeud ((NonTerminal
                             ↪  SubroutineRange, s1), l1)))
                         [] )
      | Noeud
          ( (NonTerminal SubroutineRange, _),
            [
              Noeud ((NonTerminal
              ↪  SubroutineParList_opt, s), l);
              Noeud ((Terminal EOS, s1), l1);
              Noeud ((NonTerminal BodyConstruct_star,
              ↪  s2), l2);
              Noeud ((NonTerminal EndSubroutineStmt,
              ↪  s3), l3);
            ] ) ->
          let n = ref [] in
```

```
let n2 = ref [] in
(match l with
| [ Noeud ((Terminal E, _), []) ] -> ()
| _ ->
    n :=
      [
        convert_to_abstract_aux
          (Noeud ((NonTerminal SubroutineParList_opt, s), l));
      ]);
(match l2 with
| [ Noeud ((Terminal E, _), []) ]
| [
    Noeud ((Terminal LParenthesis, _), []);
    Noeud
```

```
                ( (NonTerminal
                 ↪  SubroutinePar_Comma_SubroutinePar_star
                 ↪  _),
                  [ Noeud ((Terminal E, _), []) ] );
              Noeud ((Terminal RParenthesis, _), []);
            ] ->
              ()
          | _ ->
              n2 :=
                [
                  convert_to_abstract_aux
                    (Noeud ((NonTerminal
                     ↪  BodyConstruct_star, s2),
                     ↪  l2));
                ]);
          Noeud
            ( ToFlatten,
```

```
                flatten
                  (Noeud (ToFlatten, !n))
                  (convert_to_abstract_aux (Noeud ((Terminal EOS, s1), l1))
                  :: flatten
                      (Noeud (ToFlatten, !n2))
                      [
                        convert_to_abstract_aux
                          (Noeud ((NonTerminal EndSubroutineStmt, s3), l3));
                      ]) )
      | Noeud
          ( (NonTerminal SubroutineParList_opt, _),
            [
              Noeud ((Terminal LParenthesis, _), []);
              Noeud
```

```
                ((NonTerminal
                 ↪  SubroutinePar_Comma_SubroutinePar_sta
                 ↪  s), l);
              Noeud ((Terminal RParenthesis, _), []);
            ] ) ->
          convert_to_abstract_aux
            (Noeud ((NonTerminal
             ↪  SubroutinePar_Comma_SubroutinePar_star_op
             ↪  s), l))
      | Noeud
          ( (NonTerminal
           ↪  SubroutinePar_Comma_SubroutinePar_star_opt,
           ↪  _),
            [
              Noeud ((NonTerminal SubroutinePar, s),
               ↪  l);
              Noeud
```

```
                ( (NonTerminal Comma_SubroutinePar_star, _),
                  [ Noeud ((Terminal E, _), []) ] );
            ] ) ->
          convert_to_abstract_aux (Noeud ((NonTerminal SubroutinePar, s), l))
      | Noeud
          ( (NonTerminal SubroutinePar_Comma_SubroutinePar_star_opt, _),
            [
              Noeud ((NonTerminal SubroutinePar, s), l);
              Noeud ((NonTerminal Comma_SubroutinePar_star, s1), l1);
            ] ) ->
          Noeud
```

```
          ( ToFlatten,
            [
              convert_to_abstract_aux
                (Noeud ((NonTerminal SubroutinePar,
                ↪  s), l));
              convert_to_abstract_aux
                (Noeud ((NonTerminal
                ↪  Comma_SubroutinePar_star, s1),
                ↪  l1));
            ] )
    | Noeud
        ( (NonTerminal Comma_SubroutinePar_star,
          ↪  _),
          [
            Noeud ((Terminal Comma, _), []);
            Noeud ((NonTerminal SubroutinePar, s),
            ↪  l);
```

```
                Noeud
                  ( (NonTerminal
                  ↪  Comma_SubroutinePar_star, _),
                    [ Noeud ((Terminal E, _), []) ] );
              ] ) ->
          convert_to_abstract_aux (Noeud
           ↪  ((NonTerminal SubroutinePar, s), l))
      | Noeud
          ( (NonTerminal Comma_SubroutinePar_star,
           ↪  _),
              [
                Noeud ((Terminal Comma, _), []);
                Noeud ((NonTerminal SubroutinePar, s),
                 ↪  l);
                Noeud ((NonTerminal
                 ↪  Comma_SubroutinePar_star, s1), l1);
              ] ) ->
```

```
            Noeud
              ( ToFlatten,
                [
                  convert_to_abstract_aux
                    (Noeud ((NonTerminal SubroutinePar, s), l));
                  convert_to_abstract_aux
                    (Noeud ((NonTerminal Comma_SubroutinePar_star, s1), l1));
                ] )
      | Noeud
          ( (NonTerminal SubroutinePar, _),
            [ Noeud ((NonTerminal DummyArgName, s), l) ] ) ->
          convert_to_abstract_aux (Noeud ((NonTerminal DummyArgName, s), l))
```

```
    | Noeud
        ( (NonTerminal EndSubroutineStmt, _),
          [
            Noeud ((Terminal EndSubroutine, _), []);
            Noeud ((NonTerminal EndName_opt, _), _);
            Noeud ((Terminal EOS, s), l);
          ] ) ->
        convert_to_abstract_aux (Noeud ((Terminal EOS, s), l))
    | Noeud
        ( (NonTerminal MainProgram, _),
          [
            Noeud ((NonTerminal ProgramStmt, s), l);
```

```
                Noeud ((NonTerminal MainRange, s1),
                 ↪  l1);
              ] ) ->
            Noeud
              ( Syntax Program,
                flatten
                  (Noeud
                     ( ToFlatten,
                       [
                         convert_to_abstract_aux
                           (Noeud ((NonTerminal
                            ↪  ProgramStmt, s), l));
                         convert_to_abstract_aux
                           (Noeud ((NonTerminal
                            ↪  MainRange, s1), l1));
                       ] ))
                  [] )
```

```
      | Noeud
          ( (NonTerminal Contains_Function, _),
            [
              Noeud ((Terminal Contains, _), []);
              Noeud ((Terminal EOS, s), l);
              Noeud
                ( (NonTerminal
                ↪  FunctionSubprogram_star, _),
                  [ Noeud ((Terminal E, _), []) ] );
            ] ) ->
          convert_to_abstract_aux (Noeud ((Terminal
          ↪  EOS, s), l))
      | Noeud
          ( (NonTerminal Contains_Function, _),
            [
              Noeud ((Terminal Contains, _), []);
              Noeud ((Terminal EOS, s), l);
```

```
              Noeud ((NonTerminal FunctionSubprogram_star, s1), l1);
            ] ) ->
          Noeud
            ( ToFlatten,
              [
                convert_to_abstract_aux (Noeud ((Terminal EOS, s), l));
                convert_to_abstract_aux
                  (Noeud ((NonTerminal FunctionSubprogram_star, s1), l1));
              ] )
      | Noeud
          ( (NonTerminal FunctionSubprogram_star, _),
            [
              Noeud ((Terminal Recursive, _), []);
```

```
                    Noeud ((NonTerminal FunctionSubprogram, s), l);
                    Noeud
                      ( (NonTerminal FunctionSubprogram_star, _),
                        [ Noeud ((Terminal E, _), []) ] );
                  ] )
        | Noeud
              ( (NonTerminal FunctionSubprogram_star, _),
                [
                    Noeud ((NonTerminal FunctionSubprogram, s), l);
                    Noeud
                      ( (NonTerminal FunctionSubprogram_star, _),
                        [ Noeud ((Terminal E, _), []) ] );
                  ] ) ->
```

```
            convert_to_abstract_aux (Noeud ((NonTerminal FunctionSubprogram, s), l))
        | Noeud
            ( (NonTerminal FunctionSubprogram_star, _),
              [
                Noeud ((NonTerminal FunctionSubprogram, s), l);
                Noeud ((NonTerminal FunctionSubprogram_star, s1), l1);
              ] )
        | Noeud
            ( (NonTerminal FunctionSubprogram_star, _),
              [
                Noeud ((Terminal Recursive, _), []);
                Noeud ((NonTerminal FunctionSubprogram, s), l);
```

```
                Noeud ((NonTerminal FunctionSubprogram_star, s1), l1);
              ] ) ->
            Noeud
              ( ToFlatten,
                [
                  convert_to_abstract_aux
                    (Noeud ((NonTerminal FunctionSubprogram, s), l));
                  convert_to_abstract_aux
                    (Noeud ((NonTerminal FunctionSubprogram_star, s1), l1));
                ] )
        | Noeud
            ( (NonTerminal ProgramStmt, _),
              [
```

```
                  Noeud ((Terminal Program, _), []);
                  Noeud ((NonTerminal ProgramName, s),
                   ↪  l);
                  Noeud ((Terminal EOS, s1), l1);
                ] ) ->
            Noeud
              ( ToFlatten,
                [
                  convert_to_abstract_aux (Noeud
                   ↪  ((NonTerminal ProgramName, s),
                   ↪  l));
                  convert_to_abstract_aux (Noeud
                   ↪  ((Terminal EOS, s1), l1));
                ] )
        | Noeud
            ( (NonTerminal MainRange, _),
```

```
                [ Noeud ((NonTerminal
                ↪  Contains_Function_opt_EndProgramStmt,
                ↪  s), l) ]
              ) ->
              convert_to_abstract_aux
                (Noeud ((NonTerminal
                ↪  Contains_Function_opt_EndProgramStmt,
                ↪  s), l))
          | Noeud
              ( (NonTerminal MainRange, _),
                [
                  Noeud ((NonTerminal BodyConstruct, s),
                   ↪  l);
                  Noeud
                    ( (NonTerminal BodyConstruct_star,
                     ↪  _),
                      [ Noeud ((Terminal E, _), []) ] );
```

```
                  Noeud ((NonTerminal Contains_Function_opt_EndProgramStmt, s1), l1);
                ] ) ->
              Noeud
                ( ToFlatten,
                  [
                    convert_to_abstract_aux
                      (Noeud ((NonTerminal BodyConstruct, s), l));
                    convert_to_abstract_aux
                      (Noeud
                         ((NonTerminal Contains_Function_opt_EndProgram
                         s1), l1));
                  ] )
          | Noeud
```

```
          ( (NonTerminal MainRange, _),
            [
              Noeud ((NonTerminal BodyConstruct, s),
               ↪  l);
              Noeud ((NonTerminal BodyConstruct_star,
               ↪  s2), l2);
              Noeud ((NonTerminal
               ↪  Contains_Function_opt_EndProgramStmt,
               ↪  s1), l1);
            ] ) ->
          Noeud
            ( ToFlatten,
              [
                convert_to_abstract_aux
                  (Noeud ((NonTerminal BodyConstruct,
                   ↪  s), l));
                Noeud
```

```
                     ( ToFlatten,
                       [
                         convert_to_abstract_aux
                           (Noeud ((NonTerminal
                           ↪   BodyConstruct_star, s2),
                           ↪   l2));
                       ] );
                 convert_to_abstract_aux
                   (Noeud
                      ((NonTerminal
                      ↪   Contains_Function_opt_EndProgramS
                      ↪   s1), l1));
               ] )
      | Noeud
          ( (NonTerminal BodyConstruct_star, _),
            [
```

```
                  Noeud ((NonTerminal BodyConstruct, s),
                   ↪  l);
                  Noeud
                    ( (NonTerminal BodyConstruct_star,
                     ↪  _),
                      [ Noeud ((Terminal E, _), []) ] );
              ] ) ->
              Noeud
                ( ToFlatten,
                  [
                    convert_to_abstract_aux
                      (Noeud ((NonTerminal BodyConstruct,
                       ↪  s), l));
                  ] )
        | Noeud
            ( (NonTerminal BodyConstruct_star, _),
              [
```

```
                Noeud ((NonTerminal BodyConstruct, s), l);
                Noeud ((NonTerminal BodyConstruct_star, s1), l1);
              ] ) ->
            Noeud
              ( ToFlatten,
                [
                  convert_to_abstract_aux
                    (Noeud ((NonTerminal BodyConstruct, s), l));
                  Noeud
                    ( ToFlatten,
                      [
                        convert_to_abstract_aux
```

```
                    (Noeud ((NonTerminal BodyConstruct_star, s1), l1));
                ] );
            ] )
    | Noeud
        ( (NonTerminal Contains_Function_opt_EndProgramStmt, _),
          [ Noeud ((NonTerminal EndProgramStmt, s), l) ] ) ->
        convert_to_abstract_aux (Noeud ((NonTerminal EndProgramStmt, s), l))
    | Noeud
        ( (NonTerminal Contains_Function_opt_EndProgramStmt, _),
```

```
            [
              Noeud ((NonTerminal Contains_Function,
               ↪  s1), l1);
              Noeud ((NonTerminal EndProgramStmt, s),
               ↪  l);
            ] ) ->
          Noeud
            ( ToFlatten,
              [
                convert_to_abstract_aux
                  (Noeud ((NonTerminal
                   ↪  Contains_Function, s1), l1));
                convert_to_abstract_aux
                  (Noeud ((NonTerminal
                   ↪  EndProgramStmt, s), l));
              ] )
      | Noeud
```

```
          ( (NonTerminal EndProgramStmt, _),
            [
              Noeud ((Terminal EndProgram, _), []);
              Noeud ((NonTerminal EndName_opt, _),
               ↪  _);
              Noeud ((Terminal EOS, s), []);
            ] ) ->
          convert_to_abstract_aux (Noeud ((Terminal
          ↪  EOS, s), []))
      | Noeud
          ( (NonTerminal BodyConstruct, _),
            [ Noeud ((NonTerminal
            ↪  SpecificationPartConstruct, s), l) ]
            ↪  ) ->
          Noeud
            ( ToFlatten,
              [
```

```
                convert_to_abstract_aux
                  (Noeud ((NonTerminal SpecificationPartConstruct, s), l));
              ] )
        | Noeud
            ( (NonTerminal BodyConstruct, _),
              [ Noeud ((NonTerminal ExecutableConstruct, s), l) ] ) ->
            convert_to_abstract_aux
              (Noeud ((NonTerminal ExecutableConstruct, s), l))
        | Noeud
            ( (NonTerminal SpecificationPartConstruct, _),
              [ Noeud ((NonTerminal DeclarationConstruct, s), l) ] ) ->
```

```
          Noeud
            ( ToFlatten,
              [
                convert_to_abstract_aux
                  (Noeud ((NonTerminal DeclarationConstruct, s), l));
              ] )
      | Noeud
          ( (NonTerminal DeclarationConstruct, _),
            [ Noeud ((NonTerminal TypeDeclarationStmt, s), l) ] ) ->
          Noeud
            ( ToFlatten,
              [
                convert_to_abstract_aux
                  (Noeud ((NonTerminal TypeDeclarationStmt, s), l));
```

```
                ] )
      | Noeud
          ( (NonTerminal TypeDeclarationStmt, _),
            [
              Noeud
                ((NonTerminal TypeSpec, _), [ Noeud
                ↪  ((Terminal Double, _), []) ]);
              Noeud ((NonTerminal
              ↪  Comma_AttrSpec_star, _), l1);
              Noeud ((NonTerminal
              ↪  TypeDecl_Assignment, s), l);
              Noeud ((Terminal EOS, s2), []);
            ] ) ->
          (match l1 with
          | [ Noeud ((Terminal E, _), []) ]
          | [
              Noeud ((Terminal Comma, _), []);
```

```
                Noeud
                  ( (NonTerminal AttrSpec, _),
                    [ Noeud ((NonTerminal
                     ↪  Intent_in_out, _), _) ] );
                Noeud
                  ( (NonTerminal Comma_AttrSpec_star,
                  ↪  _),
                    [ Noeud ((Terminal E, _), []) ] );
              ] ->
                ()
            | _ -> failwith "TypeDeclarationStmt 1");
            Noeud
              ( ToFlatten,
                [
                  Noeud
                    ( Syntax Double_precision,
                      [
```

```
                        convert_to_abstract_aux
                          (Noeud ((NonTerminal
                          ↪  TypeDecl_Assignment, s),
                          ↪  l));
                      ] );
                  convert_to_abstract_aux (Noeud
                  ↪  ((Terminal EOS, s2), []));
                ] )
    | Noeud
        ( (NonTerminal TypeDeclarationStmt, _),
          [
            Noeud
              ( (NonTerminal TypeSpec, _),
                [
                  Noeud ((Terminal Integer, _),
                  ↪  []);
                  Noeud
```

```
                    ( (NonTerminal KindSelector_opt, _),
                      [ Noeud ((Terminal E, _), []) ] );
                ] );
          Noeud ((NonTerminal Comma_AttrSpec_star, _), l1);
          Noeud ((NonTerminal TypeDecl_Assignment, s), l);
          Noeud ((Terminal EOS, s2), []);
        ] ) ->
      (match l1 with
      | [ Noeud ((Terminal E, _), []) ]
      | [
          Noeud ((Terminal Comma, _), []);
          Noeud
            ( (NonTerminal AttrSpec, _),
```

```
                    [ Noeud ((NonTerminal
                     ↪  Intent_in_out, _), _) ] );
               Noeud
                 ( (NonTerminal Comma_AttrSpec_star,
                 ↪  _),
                   [ Noeud ((Terminal E, _), []) ] );
             ] ->
               ()
           | _ -> failwith "TypeDeclarationStmt 2");
           Noeud
             ( ToFlatten,
               [
                 Noeud
                   ( Syntax Integer,
                     flatten
                       (convert_to_abstract_aux
```

```
                                (Noeud ((NonTerminal TypeDecl_Assignment, s), l)))
                            [] );
                        convert_to_abstract_aux (Noeud ((Terminal EOS, s2), []));
                    ] )
        | Noeud
            ( (NonTerminal TypeDeclarationStmt, _),
              [
                Noeud
                  ( (NonTerminal TypeSpec, _),
                    [ Noeud ((Terminal Character, _), []) ] );
                Noeud ((NonTerminal Comma_AttrSpec_star, s1), l1);
```

```
            Noeud ((NonTerminal TypeDecl_Assignment, s), l);
            Noeud ((Terminal EOS, s2), []);
          ] ) ->
        let n1 = ref [] in
        (match l1 with
        | [ Noeud ((Terminal E, _), []) ]
        | [
            Noeud ((Terminal Comma, _), []);
            Noeud
              ( (NonTerminal AttrSpec, _),
                [ Noeud ((NonTerminal Intent_in_out, _), _) ] );
            Noeud
              ( (NonTerminal Comma_AttrSpec_star, _),
                [ Noeud ((Terminal E, _), []) ] );
```

```
      ] ->
        ()
    | _ ->
        n1 :=
          [
            convert_to_abstract_aux
              (Noeud ((NonTerminal Comma_AttrSpec_star, s1), l1));
          ]);
  Noeud
    ( ToFlatten,
      [
        Noeud
          ( Syntax Character,
            flatten
              (Noeud (ToFlatten, !n1))
```

```
                      (flatten
                         (convert_to_abstract_aux
                            (Noeud ((NonTerminal TypeDecl_Assignment, s), l)))
                         []) );
                convert_to_abstract_aux (Noeud ((Terminal EOS, s2), []));
            ] )
    | Noeud
        ( (NonTerminal Comma_AttrSpec_star, _),
          [
            Noeud ((NonTerminal Intent_in_out, _), _);
            Noeud ((NonTerminal Comma_AttrSpec_star, s), l);
          ] ) ->
```

```
          convert_to_abstract_aux
            (Noeud ((NonTerminal Comma_AttrSpec_star, s), l))
      | Noeud
          ( (NonTerminal Comma_AttrSpec_star, _),
            [
              Noeud ((Terminal Parameter, _), []);
              Noeud ((NonTerminal Comma_AttrSpec_star, _), _);
              (* the rest must be intent_in_out if not empty. either way, nothing to be done *)
            ] ) ->
          Noeud (Syntax Constant, [])
      | Noeud
          ( (NonTerminal TypeDeclarationStmt, _),
            [
```

```
            Noeud
              ( (NonTerminal TypeSpec, _),
                [
                  Noeud ((Terminal Complex, _), []);
                  Noeud
                    ( (NonTerminal KindSelector_opt, _),
                      [ Noeud ((Terminal E, _), []) ] );
                ] );
            Noeud ((NonTerminal Comma_AttrSpec_star, _), l1);
            Noeud ((NonTerminal TypeDecl_Assignment, s), l);
            Noeud ((Terminal EOS, s2), []);
          ] ) ->
```

```
          (match l1 with
          | [ Noeud ((Terminal E, _), []) ]
          | [
              Noeud ((Terminal Comma, _), []);
              Noeud
                ( (NonTerminal AttrSpec, _),
                  [ Noeud ((NonTerminal Intent_in_out, _), _) ] );
              Noeud
                ( (NonTerminal Comma_AttrSpec_star, _),
                  [ Noeud ((Terminal E, _), []) ] );
            ] ->
              ()
          | _ -> failwith "TypeDeclarationStmt 3");
          Noeud
            ( ToFlatten,
```

```
                [
                  Noeud
                    ( Syntax Complex,
                      flatten
                        (convert_to_abstract_aux
                          (Noeud ((NonTerminal TypeDecl_Assignment, s), l)))
                        [] );
                  convert_to_abstract_aux (Noeud ((Terminal EOS, s2), []));
                ] )
        | Noeud
            ( (NonTerminal TypeDeclarationStmt, _),
              [
                Noeud
                  ( (NonTerminal TypeSpec, _),
```

```
                [
                  Noeud ((Terminal Logical, _), [] );
                  Noeud
                    ( (NonTerminal KindSelector_opt, _),
                      [ Noeud ((Terminal E, _), []) ] );
                ] );
          Noeud ((NonTerminal Comma_AttrSpec_star, _), l1);
          Noeud ((NonTerminal TypeDecl_Assignment, s), l);
          Noeud ((Terminal EOS, s2), []);
        ] ) ->
      (match l1 with
      | [ Noeud ((Terminal E, _), []) ]
```

```
| [
    Noeud ((Terminal Comma, _), []);
    Noeud
      ( (NonTerminal AttrSpec, _),
        [ Noeud ((NonTerminal Intent_in_out, _), _) ] );
    Noeud
      ( (NonTerminal Comma_AttrSpec_star, _),
        [ Noeud ((Terminal E, _), []) ] );
  ] ->
    ()
| _ -> failwith "TypeDeclarationStmt 4");
Noeud
  ( ToFlatten,
    [
      Noeud
```

```
                  ( Syntax Logical,
                    flatten
                      (convert_to_abstract_aux
                        (Noeud ((NonTerminal TypeDecl_Assignment, s), l)))
                      [] );
                convert_to_abstract_aux (Noeud ((Terminal EOS, s2), []));
              ] )
    | Noeud
        ( (NonTerminal TypeDeclarationStmt, _),
          [
            Noeud
              ( (NonTerminal TypeSpec, _),
                [
                  Noeud ((Terminal Real, _), []);
```

```
                  Noeud
                    ( (NonTerminal
                     ↪  KindSelector_opt, _),
                      [ Noeud ((Terminal E, _), [])
                       ↪  ] );
              ] );
         Noeud ((NonTerminal
          ↪  Comma_AttrSpec_star, _), l1);
         Noeud ((NonTerminal
          ↪  TypeDecl_Assignment, s), l);
         Noeud ((Terminal EOS, s2), []);
     ] ) ->
   (match l1 with
   | [ Noeud ((Terminal E, _), []) ]
   | [
         Noeud ((Terminal Comma, _), []);
         Noeud
```

```
                  ( (NonTerminal AttrSpec, _),
                    [ Noeud ((NonTerminal Intent_in_out, _), _) ] );
                Noeud
                  ( (NonTerminal Comma_AttrSpec_star, _),
                    [ Noeud ((Terminal E, _), []) ] );
              ] ->
                ()
            | _ -> failwith "TypeDeclarationStmt 5");
            Noeud
              ( ToFlatten,
                [
                  Noeud
                    ( Syntax Real,
                      flatten
                        (convert_to_abstract_aux
```

```
                        (Noeud ((NonTerminal TypeDecl_Assignment, s), l)))
                     [] );
                 convert_to_abstract_aux (Noeud ((Terminal EOS, s2), []));
               ] )
    | Noeud
        ( (NonTerminal TypeDeclarationStmt, _),
          [
            Noeud
              ( (NonTerminal TypeSpec, _),
                [
                  Noeud ((Terminal Integer, _), []);
                  Noeud ((NonTerminal KindSelector_opt, s1), l1);
```

```
                  ] );
              Noeud ((NonTerminal
               ↪  Comma_AttrSpec_star, _), l3);
              Noeud ((NonTerminal
               ↪  TypeDecl_Assignment, s), l);
              Noeud ((Terminal EOS, s2), []);
          ] ) ->
        (match l3 with
        | [ Noeud ((Terminal E, _), []) ]
        | [
              Noeud ((Terminal Comma, _), []);
              Noeud
                ( (NonTerminal AttrSpec, _),
                  [ Noeud ((NonTerminal
                   ↪  Intent_in_out, _), _) ] );
              Noeud
```

```
                ( (NonTerminal Comma_AttrSpec_star, _),
                  [ Noeud ((Terminal E, _), []) ] );
            ] ->
              ()
        | _ -> failwith "TypeDeclarationStmt 6");
        Noeud
          ( ToFlatten,
            [
              Noeud
                ( Syntax Integer,
                  Noeud
                    ( Syntax Size,
                      [
                        convert_to_abstract_aux
                          (Noeud ((NonTerminal KindSelector_opt, s1), l1));
```

```
                         ] )
                    :: flatten
                         (convert_to_abstract_aux
                            (Noeud ((NonTerminal TypeDecl_Assignment, s), l)))
                         [] );
                 convert_to_abstract_aux (Noeud ((Terminal EOS, s2), []));
               ] )
      | Noeud
          ( (NonTerminal TypeDeclarationStmt, _),
            [
              Noeud
                ( (NonTerminal TypeSpec, _),
                  [
```

```
                    Noeud ((Terminal Complex, _), []);
                    Noeud ((NonTerminal KindSelector_opt, s1), l1);
                  ] );
              Noeud ((NonTerminal Comma_AttrSpec_star, _), l3);
              Noeud ((NonTerminal TypeDecl_Assignment, s), l);
              Noeud ((Terminal EOS, s2), []);
          ] ) ->
        (match l3 with
        | [ Noeud ((Terminal E, _), []) ]
        | [
              Noeud ((Terminal Comma, _), []);
              Noeud
                ( (NonTerminal AttrSpec, _),
```

```
                [ Noeud ((NonTerminal Intent_in_out, _), _) ] );
            Noeud
              ( (NonTerminal Comma_AttrSpec_star, _),
                [ Noeud ((Terminal E, _), []) ] );
          ] ->
            ()
        | _ -> failwith "TypeDeclarationStmt 7");
        Noeud
          ( ToFlatten,
            [
              Noeud
                ( Syntax Complex,
                  Noeud
                    ( Syntax Size,
                      [
```

```
                                        convert_to_abstract_aux
                                           (Noeud ((NonTerminal KindSelector_opt, s1), l1));
                                     ] )
                              :: flatten
                                   (convert_to_abstract_aux
                                       (Noeud ((NonTerminal TypeDecl_Assignment, s), l)))
                                     [] );
                     convert_to_abstract_aux (Noeud ((Terminal EOS, s2), []));
                  ] )
        | Noeud
             ( (NonTerminal TypeDeclarationStmt, _),
               [
```

```
                Noeud
                  ( (NonTerminal TypeSpec, _),
                    [
                      Noeud ((Terminal Logical, _), [] );
                      Noeud ((NonTerminal KindSelector_opt, s1), l1);
                    ] );
                Noeud ((NonTerminal Comma_AttrSpec_star, _), l3);
                Noeud ((NonTerminal TypeDecl_Assignment, s), l);
                Noeud ((Terminal EOS, s2), []);
              ] ) ->
            (match l3 with
            | [ Noeud ((Terminal E, _), []) ]
            | [
```

```
                  Noeud ((Terminal Comma, _), []);
                  Noeud
                    ( (NonTerminal AttrSpec, _),
                      [ Noeud ((NonTerminal
                      ↪  Intent_in_out, _), _) ] );
                  Noeud
                    ( (NonTerminal Comma_AttrSpec_star,
                    ↪  _),
                      [ Noeud ((Terminal E, _), []) ] );
                ] ->
                  ()
              | _ -> failwith "TypeDeclarationStmt 8");
              Noeud
                ( ToFlatten,
                  [
                    Noeud
                      ( Syntax Logical,
```

```
                    Noeud
                      ( Syntax Size,
                        [
                          convert_to_abstract_aux
                            (Noeud ((NonTerminal KindSelector_opt, s1), l1));
                        ] )
                    :: flatten
                         (convert_to_abstract_aux
                            (Noeud ((NonTerminal TypeDecl_Assignment, s), l)))
                         [] );
                convert_to_abstract_aux (Noeud ((Terminal EOS, s2), []));
              ] )
```

```
      | Noeud
          ( (NonTerminal TypeDeclarationStmt, _),
            [
              Noeud
                ( (NonTerminal TypeSpec, _),
                  [
                    Noeud ((Terminal Real, _), []);
                    Noeud ((NonTerminal
                     ↪  KindSelector_opt, s1), l1);
                  ] );
              Noeud ((NonTerminal
               ↪  Comma_AttrSpec_star, _), l3);
              Noeud ((NonTerminal
               ↪  TypeDecl_Assignment, s), l);
              Noeud ((Terminal EOS, s2), []);
            ] ) ->
          (match l3 with
```

```
        | [ Noeud ((Terminal E, _), []) ]
        | [
            Noeud ((Terminal Comma, _), []);
            Noeud
              ( (NonTerminal AttrSpec, _),
                [ Noeud ((NonTerminal Intent_in_out, _), _) ] );
            Noeud
              ( (NonTerminal Comma_AttrSpec_star, _),
                [ Noeud ((Terminal E, _), []) ] );
          ] ->
            ()
        | _ -> failwith "TypeDeclarationStmt 9");
        Noeud
          ( ToFlatten,
            [
```

```
Noeud
  ( Syntax Real,
    Noeud
      ( Syntax Size,
        [
          convert_to_abstract_aux
            (Noeud ((NonTerminal KindSelector_opt, s1), l1));
        ] )
    :: flatten
         (convert_to_abstract_aux
            (Noeud ((NonTerminal TypeDecl_Assignment, s), l)))
         [] );
```

```
              convert_to_abstract_aux (Noeud ((Terminal EOS, s2), []));
            ] )
      | Noeud ((NonTerminal AttrSpec, _), [ Noeud ((Terminal Parameter, _), []) ])
        ->
          Noeud (Syntax Constant, [])
      | Noeud
          ( (NonTerminal Comma_EntityDecl_star, _),
            [
              Noeud ((Terminal Comma, _), []);
              Noeud ((NonTerminal EntityDecl, s), l);
              Noeud
                ( (NonTerminal Comma_EntityDecl_star, _),
                  [ Noeud ((Terminal E, _), []) ] );
            ] )
```

```
    | Noeud
        ( (NonTerminal TypeDecl_Assignment, _),
          [
            Noeud ((Terminal Colon, _), []);
            Noeud ((Terminal Colon, _), []);
            Noeud ((NonTerminal EntityDecl, s), l);
            Noeud
              ( (NonTerminal Comma_EntityDecl_star,
              ↪  _),
                [ Noeud ((Terminal E, _), []) ] );
          ] ) ->
        convert_to_abstract_aux (Noeud
         ↪  ((NonTerminal EntityDecl, s), l))
    | Noeud
        ( (NonTerminal Comma_EntityDecl_star, _),
          [
            Noeud ((Terminal Comma, _), []);
```

```
                Noeud ((NonTerminal EntityDecl, s), l);
                Noeud ((NonTerminal Comma_EntityDecl_star, s2), l2);
              ] )
        | Noeud
            ( (NonTerminal TypeDecl_Assignment, _),
              [
                Noeud ((Terminal Colon, _), []);
                Noeud ((Terminal Colon, _), []);
                Noeud ((NonTerminal EntityDecl, s), l);
                Noeud ((NonTerminal Comma_EntityDecl_star, s2), l2);
              ] ) ->
            Noeud
              ( ToFlatten,
                [
```

```
                convert_to_abstract_aux (Noeud ((NonTerminal EntityDecl, s), l));
                Noeud
                  ( ToFlatten,
                    [
                      convert_to_abstract_aux
                        (Noeud ((NonTerminal Comma_EntityDecl_star, s2), l2));
                    ] );
              ] )
      | Noeud
          ( (NonTerminal Comma_ObjectName_star, _),
            [
              Noeud ((Terminal Comma, _), []);
              Noeud ((NonTerminal ObjectName, s), l);
```

```
                Noeud
                  ( (NonTerminal Comma_ObjectName_star, _),
                    [ Noeud ((Terminal E, _), []) ] );
              ] )
        | Noeud
            ( (NonTerminal TypeDecl_Assignment, _),
              [
                Noeud ((NonTerminal ObjectName, s), l);
                Noeud
                  ( (NonTerminal Comma_ObjectName_star, _),
                    [ Noeud ((Terminal E, _), []) ] );
              ] ) ->
            convert_to_abstract_aux (Noeud ((NonTerminal ObjectName, s), l))
        | Noeud
```

```
          ( (NonTerminal Comma_ObjectName_star, _),
            [
              Noeud ((Terminal Comma, _), []);
              Noeud ((NonTerminal ObjectName, s), l);
              Noeud ((NonTerminal Comma_ObjectName_star, s1), l1);
            ] )
      | Noeud
          ( (NonTerminal TypeDecl_Assignment, _),
            [
              Noeud ((NonTerminal ObjectName, s), l);
              Noeud ((NonTerminal Comma_ObjectName_star, s1), l1);
            ] ) ->
          Noeud
            ( ToFlatten,
              [
```

```
                convert_to_abstract_aux (Noeud ((NonTerminal ObjectName, s), l));
                convert_to_abstract_aux
                  (Noeud ((NonTerminal Comma_ObjectName_star, s1), l1));
              ] )
      | Noeud
          ( (NonTerminal EntityDecl, _),
            [
              Noeud ((NonTerminal ObjectName, s), l);
              Noeud
                ( (NonTerminal Asterisk_CharLength_opt, _),
                  [ Noeud ((Terminal E, _), []) ] );
              Noeud
```

```
                ((NonTerminal Equal_Expr_opt, _), [ Noeud ((Terminal E, _), []) ]);
            ] ) ->
          convert_to_abstract_aux (Noeud ((NonTerminal ObjectName, s), l))
      | Noeud
          ( (NonTerminal EntityDecl, _),
            [
              Noeud
                ((NonTerminal ObjectName, _), [ Noeud ((Terminal Ident, s), []) ]);
              Noeud
                ( (NonTerminal Asterisk_CharLength_opt, _),
                  [
                    Noeud ((Terminal Asterisk, _), []);
```

```
                  Noeud ((NonTerminal CharLength, s1), l1);
                ] );
              Noeud
                ((NonTerminal Equal_Expr_opt, _), [ Noeud ((Terminal E, _), []) ]);
          ] ) ->
        Noeud
          ( ToFlatten,
            [
              Noeud
                ( Syntax Size,
                  [
                    convert_to_abstract_aux
                      (Noeud ((NonTerminal CharLength, s1), l1));
                  ] );
```

```
                Noeud (Name s, []);
              ] )
      | Noeud
          ( (NonTerminal EntityDecl, _),
            [
              Noeud
                ((NonTerminal ObjectName, _), [ Noeud ((Terminal Ident, s), []) ]);
              Noeud
                ( (NonTerminal Asterisk_CharLength_opt, _),
                  [ Noeud ((Terminal E, _), []) ] );
              Noeud
                ( (NonTerminal Equal_Expr_opt, _),
                  [
                    Noeud ((Terminal Equal, _), []);
                    Noeud ((NonTerminal Expr, s2), l2);
```

```
                ] );
          ] ) ->
        Noeud
          ( Operateur Assignation,
            [
              Noeud (Name s, []);
              convert_to_abstract_aux (Noeud ((NonTerminal Expr, s2), l2));
            ] )
      | Noeud
        ( (NonTerminal EntityDecl, _),
          [
            Noeud
              ((NonTerminal ObjectName, _), [ Noeud ((Terminal Ident, s), []) ]);
            Noeud
              ( (NonTerminal Asterisk_CharLength_opt, _),
```

```
                [
                  Noeud ((Terminal Asterisk, _), []);
                  Noeud ((NonTerminal CharLength, s1), l1);
                ] );
            Noeud
              ( (NonTerminal Equal_Expr_opt, _),
                [
                  Noeud ((Terminal Equal, _), []);
                  Noeud ((NonTerminal Expr, s2), l2);
                ] );
        ] ) ->
    Noeud
      ( Operateur Assignation,
        [
```

```
                Noeud
                  ( Syntax Size,
                    [
                      convert_to_abstract_aux
                        (Noeud ((NonTerminal CharLength, s1), l1));
                    ] );
                Noeud (Name s, []);
                convert_to_abstract_aux (Noeud ((NonTerminal Expr, s2), l2));
              ] )
      | Noeud
          ( (NonTerminal CharLength, _),
            [
              Noeud ((Terminal LParenthesis, _), []);
              Noeud ((NonTerminal TypeParamValue, s), l);
```

```
                Noeud ((Terminal RParenthesis, _), []);
              ] ) ->
            convert_to_abstract_aux (Noeud ((NonTerminal TypeParamValue, s), l))
        | Noeud
            ( (NonTerminal CharLength, _),
              [ Noeud ((NonTerminal ScalarIntLiteralConstant, s), l) ] ) ->
            convert_to_abstract_aux
              (Noeud ((NonTerminal ScalarIntLiteralConstant, s), l))
        | Noeud
            ( (NonTerminal TypeParamValue, _),
              [ Noeud ((NonTerminal Expr_Or_Asterisk, s), l) ] ) ->
```

```
            convert_to_abstract_aux (Noeud ((NonTerminal Expr_Or_Asterisk, s), l))
        | Noeud
            ( (NonTerminal Expr_Or_Asterisk, _),
              [ Noeud ((Terminal Asterisk, _), []) ] ) ->
            Noeud (Syntax Any, [])
        | Noeud
            ((NonTerminal Expr_Or_Asterisk, _), [ Noeud ((NonTerminal Expr, s), l) ])
          ->
            convert_to_abstract_aux (Noeud ((NonTerminal Expr, s), l))
        | Noeud
            ( (NonTerminal KindSelector_opt, _),
              [
                Noeud ((Terminal LParenthesis, _), []);
```

```
                Noeud ((NonTerminal Expr, s), l);
                Noeud ((Terminal RParenthesis, _), []);
              ] ) ->
            convert_to_abstract_aux (Noeud ((NonTerminal Expr, s), l))
        | Noeud ((NonTerminal ExecutableConstruct, _), l) -> (
            match l with
            | [ Noeud ((NonTerminal ActionStmt, s1), l1) ] ->
                Noeud
                  ( ToFlatten,
                    [
                      convert_to_abstract_aux
                        (Noeud ((NonTerminal ActionStmt, s1), l1));
                    ] )
```

```
| [ Noeud ((NonTerminal DoConstruct, s1), l1) ] ->
    Noeud
      ( ToFlatten,
        [
          convert_to_abstract_aux
            (Noeud ((NonTerminal DoConstruct, s1), l1));
        ] )
| [ Noeud ((NonTerminal IfConstruct, s1), l1) ] ->
    Noeud
      ( ToFlatten,
        [
          convert_to_abstract_aux
            (Noeud ((NonTerminal IfConstruct, s1), l1));
```

```
                    ] )
          | [
           Noeud
             ( (NonTerminal ReturnStmt, _),
               [ Noeud ((Terminal Return, _), []);
                 ↪  Noeud ((Terminal EOS, s), l) ]
             );
          ] ->
              Noeud
                ( ToFlatten,
                    [
                      Noeud (Syntax Return, []);
                      convert_to_abstract_aux (Noeud
                       ↪  ((Terminal EOS, s), l));
                    ] )
          | _ -> failwith "ExecutableConstruct")
      | Noeud
```

```
          ( (NonTerminal ActionStmt, _),
            [ Noeud ((NonTerminal AssignmentStmt, s), l) ] ) ->
          Noeud
            ( ToFlatten,
              [
                convert_to_abstract_aux
                  (Noeud ((NonTerminal AssignmentStmt, s), l));
              ] )
      | Noeud
          ((NonTerminal ActionStmt, _), [ Noeud ((NonTerminal PrintStmt, s), l) ])
        ->
          Noeud
            ( ToFlatten,
```

```
                [ convert_to_abstract_aux (Noeud
                ↪  ((NonTerminal PrintStmt, s), l)) ]
             )
      | Noeud
          ( (NonTerminal AssignmentStmt, _),
            [
              Noeud ((NonTerminal Name, s), l);
              Noeud ((Terminal Equal, _), []);
              Noeud ((NonTerminal Expr, s1), l1);
              Noeud ((Terminal EOS, s2), []);
            ] ) ->
          Noeud
            ( ToFlatten,
              [
                Noeud
                  ( Operateur Assignation,
                    [
```

```
                    convert_to_abstract_aux (Noeud ((NonTerminal Name, s), l));
                    convert_to_abstract_aux (Noeud ((NonTerminal Expr, s1), l1));
                  ] );
              convert_to_abstract_aux (Noeud ((Terminal EOS, s2), []));
            ] )
    | Noeud
        ( (NonTerminal PrintStmt, _),
          [
            Noeud ((Terminal Print, _), []);
            Noeud ((NonTerminal FormatIdentifier, _), _);
            Noeud
```

```
                ( (NonTerminal Comma_OutputItemList_opt, _),
                  [ Noeud ((Terminal E, _), []) ] );
            Noeud ((Terminal EOS, s2), []);
          ] ) ->
        Noeud
          ( ToFlatten,
            [
              Noeud (Syntax Print, [ Noeud (Chaine "", []) ]);
              convert_to_abstract_aux (Noeud ((Terminal EOS, s2), []));
            ] )
      | Noeud
        ( (NonTerminal PrintStmt, _),
          [
            Noeud ((Terminal Print, _), []);
```

```
            Noeud ((NonTerminal FormatIdentifier, _), _);
            Noeud ((NonTerminal Comma_OutputItemList_opt, s), l);
            Noeud ((Terminal EOS, s2), []);
          ] ) ->
        Noeud
          ( ToFlatten,
            [
              Noeud
                ( Syntax Print,
                  flatten
                    (convert_to_abstract_aux
                       (Noeud ((NonTerminal Comma_OutputItemList_opt, s), l)))
                    [] );
```

```
          convert_to_abstract_aux (Noeud ((Terminal EOS, s2), []));
        ] )
| Noeud
    ( (NonTerminal Comma_OutputItemList_opt, _),
      [
        Noeud ((Terminal Comma, _), []);
        Noeud ((NonTerminal OutputItemList, s), l);
      ] ) ->
    convert_to_abstract_aux (Noeud ((NonTerminal OutputItemList, s), l))
| Noeud
    ( (NonTerminal Comma_OutputItem_star, _),
      [
        Noeud ((Terminal Comma, _), []);
```

```
            Noeud ((NonTerminal OutputItem, s), l);
            Noeud
              ( (NonTerminal Comma_OutputItem_star, _),
                [ Noeud ((Terminal E, _), []) ] );
          ] )
    | Noeud
        ( (NonTerminal OutputItemList, _),
          [
            Noeud ((NonTerminal OutputItem, s), l);
            Noeud
              ( (NonTerminal Comma_OutputItem_star, _),
                [ Noeud ((Terminal E, _), []) ] );
          ] ) ->
        convert_to_abstract_aux (Noeud ((NonTerminal OutputItem, s), l))
```

```
      | Noeud
         ( (NonTerminal Comma_OutputItem_star, _),
           [
             Noeud ((Terminal Comma, _), []);
             Noeud ((NonTerminal OutputItem, s), l);
             Noeud ((NonTerminal Comma_OutputItem_star, s1), l1);
           ] )
      | Noeud
         ( (NonTerminal OutputItemList, _),
           [
             Noeud ((NonTerminal OutputItem, s), l);
             Noeud ((NonTerminal Comma_OutputItem_star, s1), l1);
           ] ) ->
         Noeud
           ( ToFlatten,
```

```
                 [
                   convert_to_abstract_aux (Noeud ((NonTerminal OutputItem, s), l));
                   convert_to_abstract_aux
                     (Noeud ((NonTerminal Comma_OutputItem_star, s1), l1));
                 ] )
      | Noeud ((NonTerminal OutputItem, _), [ Noeud ((NonTerminal Expr, s), l) ])
        ->
          convert_to_abstract_aux (Noeud ((NonTerminal Expr, s), l))
      | Noeud
          ( (NonTerminal DoConstruct, _),
```

```
          [ Noeud ((NonTerminal BlockDoConstruct, s), l) ] ) ->
        convert_to_abstract_aux (Noeud ((NonTerminal BlockDoConstruct, s), l))
    | Noeud
        ( (NonTerminal BlockDoConstruct, _),
          [
            Noeud ((Terminal Do, _), []);
            Noeud ((NonTerminal LoopControl_opt, _), l);
            Noeud
              ( (NonTerminal ExecutionPartConstruct_star, _),
                [ Noeud ((Terminal E, _), []) ] );
            Noeud ((NonTerminal EndDoStmt, _), _);
          ] ) -> (
        match l with
```

```
            | [ Noeud ((Terminal EOS, s), []) ] ->
                Noeud
                  ( Syntax While,
                    [
                      Noeud (Booleen true, []);
                      convert_to_abstract_aux (Noeud ((Terminal EOS, s), []));
                    ] )
            | [
              Noeud ((NonTerminal LoopControl, s1), l1); Noeud ((Terminal EOS, s), []);
            ] -> (
                match l1 with
                | Noeud ((Terminal While, _), _) :: _ ->
                    Noeud
                      ( Syntax While,
```

```
                    flatten
                      (convert_to_abstract_aux
                         (Noeud ((NonTerminal
                          ↪  LoopControl, s1),
                          ↪  l1)))
                      [
                        convert_to_abstract_aux
                         ↪  (Noeud ((Terminal EOS,
                         ↪  s), []));
                      ] )
          | Noeud ((NonTerminal VariableName, _),
          ↪  _) :: _ ->
              Noeud
                ( Syntax For,
                    flatten
                      (convert_to_abstract_aux
```

```
                                (Noeud ((NonTerminal LoopControl, s1), l1)))
                              [
                                convert_to_abstract_aux (Noeud ((Terminal EOS, s), []));
                              ] )
                | _ -> failwith "BlockDoConstruct 1")
            | _ -> failwith "BlockDoConstruct 2")
    | Noeud
        ( (NonTerminal BlockDoConstruct, _),
          [
            Noeud ((Terminal Do, _), []);
            Noeud ((NonTerminal LoopControl_opt, _), l);
```

```
          Noeud ((NonTerminal ExecutionPartConstruct_star, s1), l1);
          Noeud ((NonTerminal EndDoStmt, _), _);
        ] ) -> (
      match l with
      | [ Noeud ((Terminal EOS, s), []) ] ->
          Noeud
            ( Syntax While,
              [
                Noeud (Booleen true, []);
                convert_to_abstract_aux (Noeud ((Terminal EOS, s), []));
                convert_to_abstract_aux
                  (Noeud ((NonTerminal ExecutionPartConstruct_star, s1), l1));
```

```
                ] )
        | [
         Noeud ((NonTerminal LoopControl, s2), l2);
          ↪  Noeud ((Terminal EOS, s), []);
        ] -> (
            match l2 with
            | Noeud ((Terminal While, _), _) :: _
            ↪  ->
                Noeud
                  ( Syntax While,
                    flatten
                      (Noeud
                        ( ToFlatten,
                          [

                              ↪  convert_to_abstract_au
```

```
                (Noeud ((NonTerminal LoopControl, s2), l2));
              convert_to_abstract_au
                (Noeud ((Terminal EOS, s), []));
              Noeud
                ( ToFlatten,
                  flatten
                    (convert_to_abst
                      (Noeud
                        ( ( NonTermina
                              Exec
```

```
                                                s1 ),
                                              l1 )))
                                       [] );
                               ] ))
                   [] )
        | Noeud ((NonTerminal VariableName, _),
        ↪  _) :: _ ->
          Noeud
            ( Syntax For,
              flatten
                (Noeud
                   ( ToFlatten,
                     [

                       ↪  convert_to_abstract_au
```

```
                    (Noeud ((NonTerminal
                     ↪  LoopControl,
                     ↪  s2), l2));

                  ↪  convert_to_abstract_au
                    (Noeud ((Terminal
                     ↪  EOS, s), []));
                  Noeud
                    ( ToFlatten,
                      flatten

                         ↪  (convert_to_abst
                           (Noeud
                              ( (
                              ↪  NonTermina

                                    ↪  Exec
```

```
                                                                    s1 ),
                                                                 l1 )))
                                                       [] );
                                            ] ))
                                  [] )
                     | _ -> failwith "BlockDoConstruct 3")
                | _ -> failwith "BlockDoConstruct 4")
      | Noeud
          ( (NonTerminal LoopControl, _),
            [
              Noeud ((Terminal While, _), []);
              Noeud ((Terminal LParenthesis, _), []);
              Noeud ((NonTerminal Expr, s), l);
              Noeud ((Terminal RParenthesis, _), []);
            ] ) ->
          convert_to_abstract_aux (Noeud
          ↪  ((NonTerminal Expr, s), l))
```

```
      | Noeud
          ( (NonTerminal LoopControl, _),
            [
              Noeud ((NonTerminal VariableName, s), l);
              Noeud ((Terminal Equal, _), []);
              Noeud ((NonTerminal IntRealDpExpression, s1), l1);
              Noeud ((Terminal Comma, _), []);
              Noeud ((NonTerminal IntRealDpExpression, s2), l2);
              Noeud ((NonTerminal Comma_IntRealDpExpression_opt, _), l3);
            ] ) -> (
          match l3 with
          | [ Noeud ((Terminal E, _), []) ] ->
```

```
          Noeud
            ( ToFlatten,
              [
                Noeud
                  ( Operateur Assignation,
                    [
                      convert_to_abstract_aux
                        (Noeud ((NonTerminal VariableName, s), l));
                      convert_to_abstract_aux
                        (Noeud ((NonTerminal IntRealDpExpression, s1), l1));
                    ] );
                convert_to_abstract_aux
```

```
                          (Noeud ((NonTerminal IntRealDpExpression, s2), l2));
                       Noeud (Syntax Step, [ Noeud (Integer "1", []) ]);
                     ] )
             | [
              Noeud ((Terminal Comma, _), _);
              Noeud ((NonTerminal IntRealDpExpression, s4), l4);
             ] ->
                 Noeud
                   ( ToFlatten,
                     [
                       Noeud
                         ( Operateur Assignation,
                           [
```

```
                    convert_to_abstract_aux
                      (Noeud ((NonTerminal VariableName, s), l));
                    convert_to_abstract_aux
                      (Noeud ((NonTerminal IntRealDpExpression, s1), l1));
                  ] );
            convert_to_abstract_aux
              (Noeud ((NonTerminal IntRealDpExpression, s2), l2));
            Noeud
              ( Syntax Step,
                [
                    convert_to_abstract_aux
```

```
                                (Noeud ((NonTerminal IntRealDpExpression, s4), l4));
                            ] );
                        ] )
            | _ -> failwith "LoopControl")
        | Noeud
            ( (NonTerminal IntRealDpExpression, _),
              [ Noeud ((NonTerminal Expr, s), l) ] ) ->
            convert_to_abstract_aux (Noeud ((NonTerminal Expr, s), l))
        | Noeud
            ( (NonTerminal IfConstruct, _),
              [
                Noeud ((NonTerminal IfThenStmt, _), l);
```

```
                Noeud ((NonTerminal ExecutionPartConstruct_star, s1), l1);
                Noeud
                  ((NonTerminal ElseIfStmt_ExecutionPartConstruct_sta
                  s2), l2);
                Noeud
                  ((NonTerminal ElseStmt_ExecutionPartConstruct_star_
                  s3), l3);
                Noeud
                  ( (NonTerminal EndIfStmt, _),
                    [
                      Noeud ((Terminal EndIf, _), []); Noeud ((Terminal EOS, s4), []);
```

```
                  ] );
              ] ) -> (
          let next =
            ref [ convert_to_abstract_aux (Noeud ((Terminal EOS, s4), [])) ]
          in
          let inner = ref [] in
          (match l3 with
          | [ Noeud ((Terminal E, _), []) ] -> ()
          | _ ->
              next :=
                convert_to_abstract_aux
                  (Noeud
                     ( (NonTerminal ElseStmt_ExecutionPartConstruct_
                     s3),
                       l3 ))
```

```
                :: !next);
          (match l2 with
          | [ Noeud ((Terminal E, _), []) ] -> ()
          | _ ->
              next :=
                convert_to_abstract_aux
                  (Noeud
                     ( ( NonTerminal
                     ↪   ElseIfStmt_ExecutionPartConstruct
                         s2 ),
                       l2 ))
                :: !next);
          (match l1 with
          | [ Noeud ((Terminal E, _), []) ] -> ()
          | _ ->
              inner :=
                flatten
```

```
                (convert_to_abstract_aux
                   (Noeud ((NonTerminal ExecutionPartConstruct_star, s1), l1)))
                !inner);
        match l with
        | [
         Noeud ((Terminal If, _), []);
         Noeud ((Terminal LParenthesis, _), []);
         Noeud ((NonTerminal ScalarLogicalExpr, s), l);
         Noeud ((Terminal RParenthesis, _), []);
         Noeud ((Terminal Then, _), []);
         Noeud ((Terminal EOS, s5), []);
        ] ->
            Noeud
              ( ToFlatten,
```

```
                    Noeud
                      ( Syntax If,
                        convert_to_abstract_aux
                          (Noeud ((NonTerminal ScalarLogicalExpr, s), l))
                        :: flatten
                             (convert_to_abstract_aux
                                (Noeud ((Terminal EOS, s5), [])))
                             !inner )
                    :: !next )
          | _ -> failwith "IfConstruct")
      | Noeud
          ( (NonTerminal ElseIfStmt_ExecutionPartConstruct_star_star _),
```

```
            [
              Noeud ((NonTerminal ElseIfStmt, s), l);
              Noeud ((NonTerminal ExecutionPartConstruct_star, s1), l1);
            ] ) -> (
          let inner = ref [] in
          (match l1 with
          | [ Noeud ((Terminal E, _), []) ] -> ()
          | _ ->
              inner :=
                flatten
                  (convert_to_abstract_aux
                     (Noeud ((NonTerminal ExecutionPartConstruct_star, s1), l1)))
                  !inner);
```

```
        match l with
        | [
         Noeud ((Terminal Else, _), []);
         Noeud ((Terminal If, _), []);
         Noeud ((Terminal LParenthesis, _), []);
         Noeud ((NonTerminal ScalarLogicalExpr,
          ↪  s1), l1);
         Noeud ((Terminal RParenthesis, _), []);
         Noeud ((Terminal Then, _), []);
         Noeud ((Terminal EOS, s2), []);
        ] ->
            Noeud
              ( Syntax Else_if,
                convert_to_abstract_aux
                  (Noeud ((NonTerminal
                   ↪  ScalarLogicalExpr, s1), l1))
                :: convert_to_abstract_aux (Noeud
                ↪  ((Terminal EOS, s2), []))
```

```
                    :: !inner )
            | _ -> failwith
            ↪  "ElseIfStmt_ExecutionPartConstruct_star_sta
        | Noeud
            ( (NonTerminal ExecutionPartConstruct_star,
            ↪  _),
              [
                Noeud ((NonTerminal
                ↪  ExecutionPartConstruct, s), l);
                Noeud
                  ( (NonTerminal
                  ↪  ExecutionPartConstruct_star, _),
                    [ Noeud ((Terminal E, _), []) ] );
              ] ) ->
            convert_to_abstract_aux
              (Noeud ((NonTerminal
              ↪  ExecutionPartConstruct, s), l))
```

```
      | Noeud
          ( (NonTerminal ExecutionPartConstruct_star,
          ↪  _),
            [
              Noeud ((NonTerminal
              ↪  ExecutionPartConstruct, s), l);
              Noeud ((NonTerminal
              ↪  ExecutionPartConstruct_star, s1),
              ↪  l1);
            ] ) ->
          Noeud
            ( ToFlatten,
              [
                convert_to_abstract_aux
                  (Noeud ((NonTerminal
                  ↪  ExecutionPartConstruct, s),
                  ↪  l));
```

```
                convert_to_abstract_aux
                  (Noeud ((NonTerminal
                  ↪  ExecutionPartConstruct_star,
                  ↪  s1), l1));
              ] )
      | Noeud
          ( (NonTerminal
          ↪  ElseStmt_ExecutionPartConstruct_star_opt,
          ↪  _),
            [
              Noeud ((NonTerminal ElseStmt, _), l);
              Noeud ((NonTerminal
              ↪  ExecutionPartConstruct_star, s1),
              ↪  l1);
            ] ) -> (
          let inner = ref [] in
          (match l1 with
```

```
            | [ Noeud ((Terminal E, _), []) ] -> ()
            | _ ->
                inner :=
                  convert_to_abstract_aux
                    (Noeud ((NonTerminal ExecutionPartConstruct_star, s1), l1))
                  :: !inner);
        match l with
        | [ Noeud ((Terminal Else, _), []); Noeud ((Terminal EOS, s2), []) ] ->
            Noeud
              ( Syntax Else,
                convert_to_abstract_aux (Noeud ((Terminal EOS, s2), []))
                :: !inner )
```

```
          | _ -> failwith "ElseStmt_ExecutionPartConstruct_star_opt")
      | Noeud
          ( (NonTerminal ExecutionPartConstruct, _),
            [ Noeud ((NonTerminal ExecutableConstruct, s), l) ] ) ->
          Noeud
            ( ToFlatten,
              [
                convert_to_abstract_aux
                  (Noeud ((NonTerminal ExecutableConstruct, s), l));
              ] )
      | Noeud
          ( (NonTerminal ScalarLogicalExpr, _),
            [ Noeud ((NonTerminal Expr, s), l) ] ) ->
```

```
        convert_to_abstract_aux (Noeud ((NonTerminal Expr, s), l))
    | Noeud
        ( (NonTerminal ScalarIntLiteralConstant, _),
          [ Noeud ((Terminal Icon, s), l) ] ) ->
        Noeud (Integer s, [])
    | Noeud ((NonTerminal Expr, _), [ Noeud ((NonTerminal Level5Expr, s), l) ])
      ->
        convert_to_abstract_aux (Noeud ((NonTerminal Level5Expr, s), l))
    | Noeud
        ( (NonTerminal EquivOp_EquivOperand_star, _),
          [
            Noeud ((NonTerminal EquivOp, _), _);
```

```
            Noeud ((NonTerminal EquivOperand, s), l);
            Noeud
              ( (NonTerminal EquivOp_EquivOperand_star, _),
                [ Noeud ((Terminal E, _), []) ] );
          ] )
    | Noeud
        ( (NonTerminal Level5Expr, _),
          [
            Noeud ((NonTerminal EquivOperand, s), l);
            Noeud
              ( (NonTerminal EquivOp_EquivOperand_star, _),
                [ Noeud ((Terminal E, _), []) ] );
          ] ) ->
```

```
            convert_to_abstract_aux (Noeud ((NonTerminal EquivOperand, s), l))
        | Noeud
            ( (NonTerminal EquivOp_EquivOperand_star, _),
              [
                Noeud ((NonTerminal EquivOp, _), _);
                Noeud ((NonTerminal EquivOperand, s), l);
                Noeud ((NonTerminal EquivOp_EquivOperand_star, s1), l1);
              ] )
        | Noeud
            ( (NonTerminal Level5Expr, _),
              [
```

```
          Noeud ((NonTerminal EquivOperand, s), l);
          Noeud ((NonTerminal EquivOp_EquivOperand_star, s1), l1);
        ] ) -> (
      match l1 with
      | Noeud
          ((NonTerminal EquivOp, _), [ Noeud ((Terminal Equivalent, _), []) ])
        :: _ ->
          Noeud
            ( OperateurLogique Equivalent,
              [
                convert_to_abstract_aux
                  (Noeud ((NonTerminal EquivOperand, s), l));
```

```
                    convert_to_abstract_aux
                      (Noeud ((NonTerminal EquivOp_EquivOperand_star, s1), l1));
                ] )
        | Noeud
            ( (NonTerminal EquivOp, _),
              [ Noeud ((Terminal NotEquivalent, _), []) ] )
          :: _ ->
            Noeud
              ( OperateurLogique NonEquivalent,
                [
                  convert_to_abstract_aux
                    (Noeud ((NonTerminal EquivOperand, s), l));
                  convert_to_abstract_aux
```

```
                    (Noeud ((NonTerminal EquivOp_EquivOperand_star, s1), l1));
                ] )
        | _ -> failwith "Level5Expr")
    | Noeud
        ( (NonTerminal OrOp_OrOperand_star, _),
          [
            Noeud ((Terminal OrOp, _), []);
            Noeud ((NonTerminal OrOperand, s), l);
            Noeud
              ( (NonTerminal OrOp_OrOperand_star, _),
                [ Noeud ((Terminal E, _), []) ] );
          ] )
    | Noeud
        ( (NonTerminal EquivOperand, _),
```

```
                [
                  Noeud ((NonTerminal OrOperand, s), l);
                  Noeud
                    ( (NonTerminal OrOp_OrOperand_star, _),
                      [ Noeud ((Terminal E, _), []) ] );
                ] ) ->
            convert_to_abstract_aux (Noeud ((NonTerminal OrOperand, s), l))
        | Noeud
            ( (NonTerminal OrOp_OrOperand_star, _),
              [
                Noeud ((Terminal OrOp, _), []);
                Noeud ((NonTerminal OrOperand, s), l);
                Noeud ((NonTerminal OrOp_OrOperand_star, s2), l2);
              ] )
```

```
    | Noeud
        ( (NonTerminal EquivOperand, _),
          [
            Noeud ((NonTerminal OrOperand, s), l);
            Noeud ((NonTerminal OrOp_OrOperand_star, s2), l2);
          ] ) ->
        Noeud
          ( OperateurLogique Ou,
            [
              convert_to_abstract_aux (Noeud ((NonTerminal OrOperand, s), l));
              convert_to_abstract_aux
                (Noeud ((NonTerminal OrOp_OrOperand_star, s2), l2));
            ] )
    | Noeud
```

```
           ( (NonTerminal AndOp_AndOperand_star, _),
             [
               Noeud ((Terminal AndOp, _), []);
               Noeud ((NonTerminal AndOperand, s), l);
               Noeud
                 ( (NonTerminal AndOp_AndOperand_star,
                 ↪  _),
                   [ Noeud ((Terminal E, _), []) ] );
             ] )
       | Noeud
           ( (NonTerminal OrOperand, _),
             [
               Noeud ((NonTerminal AndOperand, s), l);
               Noeud
                 ( (NonTerminal AndOp_AndOperand_star,
                 ↪  _),
                   [ Noeud ((Terminal E, _), []) ] );
```

```
                ] ) ->
              convert_to_abstract_aux (Noeud ((NonTerminal AndOperand, s), l))
          | Noeud
              ( (NonTerminal AndOp_AndOperand_star, _),
                [
                  Noeud ((Terminal AndOp, _), []);
                  Noeud ((NonTerminal AndOperand, s1), l1);
                  Noeud ((NonTerminal AndOp_AndOperand_star, s2), l2);
                ] )
          | Noeud
              ( (NonTerminal OrOperand, _),
                [
                  Noeud ((NonTerminal AndOperand, s1), l1);
```

```
                Noeud ((NonTerminal AndOp_AndOperand_star, s2), l2);
              ] ) ->
          Noeud
            ( OperateurLogique Et,
              [
                convert_to_abstract_aux (Noeud ((NonTerminal AndOperand, s1), l1));
                convert_to_abstract_aux
                  (Noeud ((NonTerminal AndOp_AndOperand_star, s2), l2));
              ] )
      | Noeud
          ( (NonTerminal AndOperand, _),
            [
```

```
                Noeud ((NonTerminal NotOp_opt, _), [ Noeud ((Terminal E, _), []) ]);
                Noeud ((NonTerminal Level4Expr, s), l);
              ] ) ->
            convert_to_abstract_aux (Noeud ((NonTerminal Level4Expr, s), l))
        | Noeud
            ( (NonTerminal AndOperand, _),
              [
                Noeud
                  ((NonTerminal NotOp_opt, _), [ Noeud ((Terminal NotOp, _), []) ]);
                Noeud ((NonTerminal Level4Expr, s), l);
              ] ) ->
            Noeud
              ( OperateurLogique Non,
```

```
              [ convert_to_abstract_aux (Noeud ((NonTerminal Level4Expr, s), l)) ]
            )
      | Noeud
          ( (NonTerminal RelOp_Level3Expr_star, _),
            [
              Noeud ((NonTerminal RelOp, _), _);
              Noeud ((NonTerminal Level3Expr, s), l);
              Noeud
                ( (NonTerminal RelOp_Level3Expr_star, _),
                  [ Noeud ((Terminal E, _), []) ] );
            ] )
      | Noeud
          ( (NonTerminal Level4Expr, _),
            [
              Noeud ((NonTerminal Level3Expr, s), l);
```

```
                  Noeud
                    ( (NonTerminal RelOp_Level3Expr_star,
                    ↪  _),
                      [ Noeud ((Terminal E, _), []) ] );
                ] ) ->
            convert_to_abstract_aux (Noeud
            ↪  ((NonTerminal Level3Expr, s), l))
        | Noeud
            ( (NonTerminal RelOp_Level3Expr_star, _),
              [
                  Noeud ((NonTerminal RelOp, _), _);
                  Noeud ((NonTerminal Level3Expr, s), l);
                  Noeud ((NonTerminal
                  ↪  RelOp_Level3Expr_star, s1), l1);
              ] )
        | Noeud
            ( (NonTerminal Level4Expr, _),
```

```
          [
            Noeud ((NonTerminal Level3Expr, s), l);
            Noeud ((NonTerminal RelOp_Level3Expr_star, s1), l1);
          ] ) -> (
        match l1 with
        | [ Noeud ((Terminal E, _), []) ] ->
            convert_to_abstract_aux (Noeud ((NonTerminal Level3Expr, s), l))
        | Noeud ((NonTerminal RelOp, _), [ Noeud ((Terminal IsEqual, _), []) ])
          :: _ ->
            Noeud
              ( Comparateur Egal,
                [
                  convert_to_abstract_aux
```

```
                        (Noeud ((NonTerminal Level3Expr, s), l));
                    convert_to_abstract_aux
                        (Noeud ((NonTerminal RelOp_Level3Expr_star, s1), l1));
                  ] )
        | Noeud ((NonTerminal RelOp, _), [ Noeud ((Terminal NotEqual, _), []) ])
          :: _ ->
            Noeud
              ( Comparateur NonEgal,
                [
                  convert_to_abstract_aux
                    (Noeud ((NonTerminal Level3Expr, s), l));
                  convert_to_abstract_aux
```

```
                        (Noeud ((NonTerminal
                         ↪  RelOp_Level3Expr_star, s1),
                         ↪  l1));
                   ] )
          | Noeud
              ((NonTerminal RelOp, _), [ Noeud
              ↪  ((Terminal StrictLess, _), []) ])
            :: _ ->
              Noeud
                ( Comparateur StrictPlusPetit,
                  [
                    convert_to_abstract_aux
                      (Noeud ((NonTerminal
                       ↪  Level3Expr, s), l));
                    convert_to_abstract_aux
```

```
                    (Noeud ((NonTerminal RelOp_Level3Expr_star, s1), l1));
                ] )
        | Noeud ((NonTerminal RelOp, _), [ Noeud ((Terminal LessEqual, _), []) ])
          :: _ ->
            Noeud
              ( Comparateur PlusPetit,
                [
                  convert_to_abstract_aux
                    (Noeud ((NonTerminal Level3Expr, s), l));
                  convert_to_abstract_aux
                    (Noeud ((NonTerminal RelOp_Level3Expr_star, s1), l1));
```

```
                    ] )
            | Noeud
                ((NonTerminal RelOp, _), [ Noeud
                ↪   ((Terminal StrictGreater, _), [])
                ↪   ])
              :: _ ->
                Noeud
                  ( Comparateur StrictPlusGrand,
                    [
                      convert_to_abstract_aux
                        (Noeud ((NonTerminal
                        ↪   Level3Expr, s), l));
                      convert_to_abstract_aux
                        (Noeud ((NonTerminal
                        ↪   RelOp_Level3Expr_star, s1),
                        ↪   l1));
                    ] )
```

```
              | Noeud
                  ((NonTerminal RelOp, _), [ Noeud ((Terminal GreaterEqual, _), []) ])
                :: _ ->
                  Noeud
                    ( Comparateur PlusGrand,
                      [
                        convert_to_abstract_aux
                          (Noeud ((NonTerminal Level3Expr, s), l));
                        convert_to_abstract_aux
                          (Noeud ((NonTerminal RelOp_Level3Expr_star, s1), l1));
                      ] )
              | _ -> failwith "Level4Expr")
          | Noeud
```

```
            ((NonTerminal Level3Expr, _), [ Noeud
             ↪  ((NonTerminal Level2Expr, s), l) ])
          ->
            convert_to_abstract_aux (Noeud
             ↪  ((NonTerminal Level2Expr, s), l))
        | Noeud
            ( (NonTerminal
             ↪  AddOp_Sign_opt_AddOperand_star, _),
              [
                Noeud ((NonTerminal AddOp, _), _);
                Noeud ((NonTerminal
                 ↪  Sign_opt_AddOperand, s), l);
                Noeud
                  ( (NonTerminal
                  ↪  AddOp_Sign_opt_AddOperand_star,
                  ↪  _),
                    [ Noeud ((Terminal E, _), []) ] );
```

```
                ] )
        | Noeud
            ( (NonTerminal Level2Expr, _),
              [
                Noeud ((NonTerminal
                 ↪  Sign_opt_AddOperand, s), l);
                Noeud
                  ( (NonTerminal
                   ↪  AddOp_Sign_opt_AddOperand_star,
                   ↪  _),
                    [ Noeud ((Terminal E, _), []) ] );
              ] ) ->
            convert_to_abstract_aux
              (Noeud ((NonTerminal Sign_opt_AddOperand,
               ↪  s), l))
        | Noeud
```

```
            ( (NonTerminal
            ↪  AddOp_Sign_opt_AddOperand_star, _),
              [
                Noeud ((NonTerminal AddOp, _), _);
                Noeud ((NonTerminal
                 ↪  Sign_opt_AddOperand, s), l);
                Noeud ((NonTerminal
                 ↪  AddOp_Sign_opt_AddOperand_star,
                 ↪  s2), l2);
              ] )
        | Noeud
            ( (NonTerminal Level2Expr, _),
              [
                Noeud ((NonTerminal
                 ↪  Sign_opt_AddOperand, s), l);
```

```
            Noeud ((NonTerminal AddOp_Sign_opt_AddOperand_star, s2), l2);
          ] ) -> (
        match l2 with
        | Noeud
            ( (NonTerminal AddOp, _),
              [
                Noeud ((NonTerminal Sign, _), [ Noeud ((Terminal Plus, _), []) ]);
              ] )
          :: _ ->
            Noeud
              ( Operateur Plus,
                [
                  convert_to_abstract_aux
```

```
                    (Noeud ((NonTerminal Sign_opt_AddOperand, s), l));
                  convert_to_abstract_aux
                    (Noeud ((NonTerminal AddOp_Sign_opt_AddOperand_star, s2), l2));
                ] )
        | Noeud
            ( (NonTerminal AddOp, _),
              [
                Noeud
                  ((NonTerminal Sign, _), [ Noeud ((Terminal Minus, _), []) ]);
              ] )
          :: _ ->
            Noeud
```

```
                  ( Operateur Moins,
                    [
                      convert_to_abstract_aux
                        (Noeud ((NonTerminal Sign_opt_AddOperand, s), l));
                      convert_to_abstract_aux
                        (Noeud ((NonTerminal AddOp_Sign_opt_AddOperand_star, s2), l2));
                    ] )
              | _ -> failwith "Level2Expr")
      | Noeud
          ( (NonTerminal Sign_opt_AddOperand, _),
            [
              Noeud ((NonTerminal Sign_opt, _), [ Noeud ((Terminal E, _), []) ]);
```

```
                Noeud ((NonTerminal AddOperand, s), l);
              ] ) ->
            convert_to_abstract_aux (Noeud ((NonTerminal AddOperand, s), l))
        | Noeud
            ( (NonTerminal Sign_opt_AddOperand, _),
              [
                Noeud
                  ( (NonTerminal Sign_opt, _),
                    [
                      Noeud
                        ((NonTerminal Sign, _), [ Noeud ((Terminal Plus, _), []) ]);
                    ] );
                Noeud ((NonTerminal AddOperand, s), l);
              ] ) ->
```

```
        Noeud
          ( Operateur Plus,
            [ convert_to_abstract_aux (Noeud ((NonTerminal AddOperand, s), l)) ]
          )
    | Noeud
        ( (NonTerminal Sign_opt_AddOperand, _),
          [
            Noeud
              ( (NonTerminal Sign_opt, _),
                [
                  Noeud
                    ((NonTerminal Sign, _), [ Noeud ((Terminal Minus, _), []) ]);
                ] );
            Noeud ((NonTerminal AddOperand, s), l);
```

```
            ] ) ->
          Noeud
            ( Operateur Moins,
              [ convert_to_abstract_aux (Noeud ((NonTerminal AddOperand, s), l)) ]
            )
      | Noeud
          ( (NonTerminal MultOp_MultOperand_star, _),
            [
              Noeud ((NonTerminal MultOp, _), _);
              Noeud ((NonTerminal MultOperand, s), l);
              Noeud
                ( (NonTerminal MultOp_MultOperand_star, _),
                  [ Noeud ((Terminal E, _), []) ] );
            ] )
```

```
      | Noeud
          ( (NonTerminal AddOperand, _),
            [
              Noeud ((NonTerminal MultOperand, s), l);
              Noeud
                ( (NonTerminal MultOp_MultOperand_star, _),
                  [ Noeud ((Terminal E, _), []) ] );
            ] ) ->
          convert_to_abstract_aux (Noeud ((NonTerminal MultOperand, s), l))
      | Noeud
          ( (NonTerminal MultOp_MultOperand_star, _),
            [
              Noeud ((NonTerminal MultOp, _), _);
```

```
                Noeud ((NonTerminal MultOperand, s), l);
                Noeud ((NonTerminal MultOp_MultOperand_star, s2), l2);
              ] )
      | Noeud
          ( (NonTerminal AddOperand, _),
            [
                Noeud ((NonTerminal MultOperand, s), l);
                Noeud ((NonTerminal MultOp_MultOperand_star, s2), l2);
              ] ) -> (
          match l2 with
          | Noeud ((NonTerminal MultOp, _), [ Noeud ((Terminal Asterisk, _), []) ])
            :: _ ->
```

```
                Noeud
                  ( Operateur Fois,
                    [
                      convert_to_abstract_aux
                        (Noeud ((NonTerminal
                        ↪  MultOperand, s), l));
                      convert_to_abstract_aux
                        (Noeud ((NonTerminal
                        ↪  MultOp_MultOperand_star,
                        ↪  s2), l2));
                    ] )
            | Noeud ((NonTerminal MultOp, _), [ Noeud
            ↪  ((Terminal Divise, _), []) ])
              :: _ ->
                Noeud
                  ( Operateur Division,
                    [
```

```
                        convert_to_abstract_aux
                          (Noeud ((NonTerminal MultOperand, s), l));
                        convert_to_abstract_aux
                          (Noeud ((NonTerminal MultOp_MultOperand_star, s2), l2));
                    ] )
            | _ -> failwith "AddOperand")
        | Noeud
            ( (NonTerminal PowerOp_Level1Expr_star, _),
              [
                Noeud ((Terminal PowerOp, _), []);
                Noeud ((NonTerminal Level1Expr, s), l);
                Noeud
                  ( (NonTerminal PowerOp_Level1Expr_star, _),
```

```
                [ Noeud ((Terminal E, _), []) ] );
            ] )
      | Noeud
          ( (NonTerminal MultOperand, _),
            [
              Noeud ((NonTerminal Level1Expr, s), l);
              Noeud
                ( (NonTerminal
                ↪  PowerOp_Level1Expr_star, _),
                [ Noeud ((Terminal E, _), []) ] );
            ] ) ->
          convert_to_abstract_aux (Noeud
          ↪  ((NonTerminal Level1Expr, s), l))
      | Noeud
          ( (NonTerminal PowerOp_Level1Expr_star, _),
            [
              Noeud ((Terminal PowerOp, _), []);
```

```
                Noeud ((NonTerminal Level1Expr, s), l);
                Noeud ((NonTerminal
                 ↪  PowerOp_Level1Expr_star, s2), l2);
              ] )
        | Noeud
            ( (NonTerminal MultOperand, _),
              [
                Noeud ((NonTerminal Level1Expr, s), l);
                Noeud ((NonTerminal
                 ↪  PowerOp_Level1Expr_star, s2), l2);
              ] ) ->
            Noeud
              ( Operateur Puissance,
                [
                  convert_to_abstract_aux (Noeud
                   ↪  ((NonTerminal Level1Expr, s),
                   ↪  l));
```

```
                    convert_to_abstract_aux
                      (Noeud ((NonTerminal PowerOp_Level1Expr_star, s2), l2));
                ] )
        | Noeud
            ((NonTerminal Level1Expr, _), [ Noeud ((NonTerminal Primary, s), l) ])
          ->
            convert_to_abstract_aux (Noeud ((NonTerminal Primary, s), l))
        | Noeud ((NonTerminal Primary, _), l) -> (
            match l with
            | [ Noeud ((Terminal Icon, s), []) ] -> Noeud (Integer s, [])
            | [ Noeud ((Terminal Rcon, s), []) ] -> Noeud (Floating s, [])
```

```
          | [ Noeud ((Terminal Dcon, s), []) ] ->
          ↪  Noeud (Double s, [])
          | [
           Noeud ((NonTerminal Name, _), [ Noeud
            ↪  ((Terminal Ident, s), []) ]);
           Noeud
             ( (NonTerminal FunctionReference_opt,
              ↪  _),
               [ Noeud ((Terminal E, _), []) ] );
          ] ->
             Noeud (Name s, [])
          | [
           Noeud ((NonTerminal Name, _), [ Noeud
            ↪  ((Terminal Ident, s), []) ]);
           Noeud
             ( (NonTerminal FunctionReference_opt,
              ↪  _),
```

```
                [
                  Noeud ((Terminal LParenthesis, _), []);
                  Noeud
                    ( ( NonTerminal
                          FunctionArg_Comma_FunctionArg
                        _ ),
                      [ Noeud ((Terminal RParenthesis, _), []) ] );
                ] );
          ] ->
            Noeud (Syntax Call, [ Noeud (Name s, []) ])
          | [
              Noeud ((NonTerminal Name, _), [ Noeud ((Terminal Ident, s), []) ]);
```

```
        Noeud ((NonTerminal FunctionReference_opt, s1), l1);
      ] ->
          Noeud
            ( Syntax Call,
              Noeud (Name s, [])
              :: flatten
                   (convert_to_abstract_aux
                      (Noeud ((NonTerminal FunctionReference_opt, s1), l1)))
                   [] )
      | [
        Noeud ((NonTerminal Scon, _), [ Noeud ((Terminal SconSingle, s), []) ]);
      ] ->
          Noeud (Chaine s, [])
```

```
        | [
         Noeud ((NonTerminal Scon, _), [ Noeud ((Terminal SconDouble, s), []) ]);
        ] ->
            Noeud (Chaine s, [])
        | [
         Noeud
           ((NonTerminal LogicalConstant, _), [ Noeud ((Terminal True, _), []) ]);
        ] ->
            Noeud (Booleen true, [])
        | [
         Noeud
           ( (NonTerminal LogicalConstant, _),
             [ Noeud ((Terminal False, _), []) ] );
        ] ->
            Noeud (Booleen false, [])
```

```
          | [
           Noeud ((Terminal LParenthesis, _), []);
           Noeud ((NonTerminal Expr, s), l);
           Noeud ((Terminal RParenthesis, _), []);
          ] ->
              Noeud
                ( ToFlatten,
                  [
                    Noeud (Parentheseouvrante, []);
                    convert_to_abstract_aux (Noeud
                     ↪  ((NonTerminal Expr, s), l));
                    Noeud (Parenthesefermante, []);
                  ] )
          | _ -> failwith "Primary")
      | Noeud
          ( (NonTerminal FunctionReference_opt, _),
            [
```

```
            Noeud ((Terminal LParenthesis, _), []);
            Noeud
              ( ( NonTerminal
              ↪  FunctionArg_Comma_FunctionArg_star_opt
                  s ),
                l );
          ] ) ->
        convert_to_abstract_aux
          (Noeud
             ( ( NonTerminal
             ↪  FunctionArg_Comma_FunctionArg_star_opt_
                 s ),
               l ))
    | Noeud
        ( (NonTerminal
        ↪  FunctionArg_Comma_FunctionArg_star_opt_RPare
        ↪  _),
```

```
                [
                  Noeud ((NonTerminal FunctionArg, s),
                   ↪  l);
                  Noeud
                    ( (NonTerminal
                     ↪  Comma_FunctionArg_star, _),
                      [ Noeud ((Terminal E, _), []) ] );
                  Noeud ((Terminal RParenthesis, _), []);
                ] ) ->
            convert_to_abstract_aux (Noeud
             ↪  ((NonTerminal FunctionArg, s), l))
        | Noeud
            ( (NonTerminal
             ↪  FunctionArg_Comma_FunctionArg_star_opt_RPare
             ↪  _),
              [
```

```
                Noeud ((NonTerminal FunctionArg, s), l);
                Noeud ((NonTerminal Comma_FunctionArg_star, s1), l1);
                Noeud ((Terminal RParenthesis, _), []);
              ] ) ->
            Noeud
              ( ToFlatten,
                [
                  convert_to_abstract_aux (Noeud ((NonTerminal FunctionArg, s), l));
                  convert_to_abstract_aux
                    (Noeud ((NonTerminal Comma_FunctionArg_star, s1), l1));
                ] )
```

```
      | Noeud
          ( (NonTerminal Comma_FunctionArg_star, _),
            [
              Noeud ((Terminal Comma, _), []);
              Noeud ((NonTerminal FunctionArg, s), l);
              Noeud
                ( (NonTerminal Comma_FunctionArg_star, _),
                  [ Noeud ((Terminal E, _), []) ] );
            ] ) ->
          convert_to_abstract_aux (Noeud ((NonTerminal FunctionArg, s), l))
      | Noeud
          ( (NonTerminal Comma_FunctionArg_star, _),
            [
              Noeud ((Terminal Comma, _), []);
```

```
          Noeud ((NonTerminal FunctionArg, s), l);
          Noeud ((NonTerminal Comma_FunctionArg_star, s1), l1);
        ] ) ->
      Noeud
        ( ToFlatten,
          [
            convert_to_abstract_aux (Noeud ((NonTerminal FunctionArg, s), l));
            convert_to_abstract_aux
              (Noeud ((NonTerminal Comma_FunctionArg_star, s1), l1));
          ] )
```

```
    | Noeud ((NonTerminal FunctionArg, _), [ Noeud
    ↪  ((NonTerminal Expr, s), l) ])
      ->
        convert_to_abstract_aux (Noeud
         ↪  ((NonTerminal Expr, s), l))
    | Noeud ((NonTerminal Name, _), [ Noeud
    ↪  ((Terminal Ident, s), []) ])
    | Noeud ((NonTerminal ArrayName, _), [ Noeud
    ↪  ((Terminal Ident, s), []) ])
    | Noeud ((NonTerminal ComponentName, _), [
    ↪  Noeud ((Terminal Ident, s), []) ])
    | Noeud ((NonTerminal EndName, _), [ Noeud
    ↪  ((Terminal Ident, s), []) ])
    | Noeud ((NonTerminal DummyArgName, _), [ Noeud
    ↪  ((Terminal Ident, s), []) ])
    | Noeud ((NonTerminal FunctionName, _), [ Noeud
    ↪  ((Terminal Ident, s), []) ])
```

```
    | Noeud
        ((NonTerminal ImpliedDoVariable, _), [ Noeud ((Terminal Ident, s), []) ])
    | Noeud ((NonTerminal ProgramName, _), [ Noeud ((Terminal Ident, s), []) ])
    | Noeud
        ((NonTerminal SubroutineName, _), [ Noeud ((Terminal Ident, s), []) ])
    | Noeud
        ((NonTerminal SubroutineNameUse, _), [ Noeud ((Terminal Ident, s), []) ])
    | Noeud ((NonTerminal VariableName, _), [ Noeud ((Terminal Ident, s), []) ])
    | Noeud ((NonTerminal ObjectName, _), [ Noeud ((Terminal Ident, s), []) ])
      ->
       Noeud (Name s, [])
```

```
    | Noeud ((Terminal EOS, s), []) ->
      Noeud (ToFlatten, convert_comments (Noeud (Commentaire s, [])))
    | _ ->
      (let string_of_symbol (s : symbol) : string =
         match s with
         | Terminal x -> string_of_terminal x
         | NonTerminal x -> string_of_non_terminal x
       in
       match t with
       | Noeud ((s, _), Noeud ((x, _), _) :: _) ->
           prerr_string (string_of_symbol s);
           prerr_char ' ';
           prerr_string (string_of_symbol x);
```

```
                prerr_newline ()
            | Noeud ((s, _), _) ->
                prerr_string (string_of_symbol s);
                prerr_newline ());
          failwith "is not implemented yet"
    in
    link_return_function (convert_to_abstract_aux t)
    ↪  None
```

```
open Automates

let syntax_automate_det = {
```

```
    nodes = [244; 243; 242; 241; 240; 239; 238; 237;
     ↪  236; 235; 234; 233; 232; 231; 230; 229; 228;
     ↪  227; 226; 225; 224; 223; 222; 221; 220; 219;
     ↪  218; 217; 216; 215; 214; 213; 212; 211; 210;
     ↪  209; 208; 207; 206; 205; 204; 203; 202; 201;
     ↪  200; 199; 198; 197; 196; 195; 194; 193; 192;
     ↪  191; 190; 189; 188; 187; 186; 185; 184; 183;
     ↪  182; 181; 180; 179; 178; 177; 176; 175; 174;
     ↪  173; 172; 171; 170; 169; 168; 167; 166; 165;
     ↪  164; 163; 162; 161; 160; 159; 158; 157; 156;
     ↪  155; 154; 153; 152; 151; 150; 149; 148; 147;
     ↪  146; 145; 144; 143; 142; 141; 140; 139; 138;
     ↪  137; 136; 135; 134; 133; 132; 131; 130; 129;
     ↪  128; 127; 126; 125; 124; 123; 122; 121; 120;
     ↪  119; 118; 117; 116; 115; 114; 113; 112; 111;
     ↪  110; 109; 108; 107; 106; 105; 104; 103; 102;
     ↪  101; 100; 99; 98; 97; 96; 95; 94; 93; 92; 91;
     ↪  90; 89; 88; 87; 86; 85; 84; 83; 82; 81; 80;
```

```
    debut = 0;
```

```
fin = [|None; Some EOS; Some Space; None; None;
    None; Some LParenthesis; Some RParenthesis;
    Some Asterisk; Some Plus; Some Comma; Some
    Minus; None; Some Divise; Some Icon; Some
    Colon; Some StrictLess; Some Equal; Some
    StrictGreater; Some Ident; Some Ident; Some
    Ident; Some Ident; Some Ident; Some Ident;
    Some Ident; Some Ident; Some Ident; Some
    Ident; Some Ident; Some Ident; Some Ident;
    Some Ident; Some Ident; Some Ident; Some
    While; Some Ident; Some Ident; Some Then;
    Some Ident; Some Ident; Some Ident; Some
    Ident; Some Ident; Some Ident; Some Ident;
    Some Ident; Some Subroutine; Some Ident; Some
    Ident; Some Ident; Some Ident; Some Ident;
    Some Ident; Some Ident; Some Return; Some
    Ident; Some Ident; Some Result; Some Ident;
    Some Ident; Some Ident; Some Ident; Some
```

```
  transitions = [|
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; 1;
     ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; 2;
     ↪ 3; 4; -1; -1; -1; -1; 5; 6; 7; 8; 9; 10;
     ↪ 11; 12; 13; 14; 14; 14; 14; 14; 14; 14; 14;
     ↪ 14; 14; 15; -1; 16; 17; 18; -1; -1; 19; 19;
     ↪ 20; 21; 22; 23; 19; 19; 24; 19; 19; 25; 19;
     ↪ 19; 26; 27; 19; 28; 29; 30; 19; 19; 31; 19;
     ↪ 19; 19; -1; -1; -1; -1; -1; -1; 19; 19; 20;
     ↪ 21; 22; 23; 19; 19; 24; 19; 19; 25; 19; 19;
     ↪ 26; 27; 19; 28; 29; 30; 19; 19; 31; 19; 19;
     ↪ 19; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; 1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  244; 3; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 244; 3; 3; 3;
    ↪  3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3;
    ↪  3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3;
    ↪  3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3;
    ↪  3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3;
    ↪  3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3;
    ↪  3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3;
    ↪  3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3;
    ↪  3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3;
    ↪  3; 3|];
```

```
[|4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4;
↪  4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4;
↪  4; 4; 4; 4; 4; 243; 4; 4; 4; 4; 4; 4; 4; 4;
↪  4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4;
↪  4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4;
↪  4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4;
↪  4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4;
↪  4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4;
↪  4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4; 4;
↪  4|];
```

```
    [|5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5;
    ↪  5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5;
    ↪  5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 242; 5; 5; 5;
    ↪  5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5;
    ↪  5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5;
    ↪  5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5;
    ↪  5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5;
    ↪  5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5;
    ↪  5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5; 5;
    ↪  5|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; 241;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; 202; 202; 202; 202; 202;
↪  202; 202; 202; 202; 202; -1; -1; -1; -1;
↪  -1; -1; -1; 203; -1; -1; -1; 204; 205; 206;
↪  -1; -1; -1; -1; 207; -1; 208; 209; -1; -1;
↪  -1; -1; 210; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; 203; -1; -1; -1; 204;
↪  205; 206; -1; -1; -1; -1; 207; -1; 208;
↪  209; -1; -1; -1; -1; 210; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; 201; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; 186; -1; 187; 187; 187; 187; 187;
↪  187; 187; 187; 187; 187; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; 188; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; 185; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; 184; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; 183; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1;
       ↪  160; 19; 19; 19; 19; 19; 19; 161; 19; 19;
       ↪  19; 19; 19; 19; 162; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; -1; -1; -1; -1; 19;
       ↪  -1; 160; 19; 19; 19; 19; 19; 19; 161; 19;
       ↪  19; 19; 19; 19; 19; 162; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; -1; -1; -1; -1;
       ↪  -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
↪ 19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
↪ 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪ 19; 19; 145; 19; 19; 19; 19; 19; 19; 19;
↪ 19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
↪ 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪ 19; 19; 145; 19; 19; 19; 19; 19; 19; 19;
↪ 19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  110; 19; 111; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; -1; -1; -1; -1; 19; -1;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  110; 19; 111; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 103; 19;
    ↪  19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 103; 19;
    ↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
↪ 19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
↪ 19; 19; 19; 19; 91; 19; 19; 19; 19; 19; 19;
↪ 19; 92; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪ 19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
↪ 19; 19; 19; 91; 19; 19; 19; 19; 19; 19; 19;
↪ 92; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪ 19; 19; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 85; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 85; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 83; 19; 19;
        ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 83; 19; 19; 19;
        ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 66;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 67; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 66; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 67; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 48; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 48; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 39; 19; 19;
        ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 39; 19; 19; 19;
        ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
       ↪  19; 19; 19; 19; 19; 19; 36; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
       ↪  19; 19; 19; 19; 19; 36; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
        ↪  19; 19; 19; 19; 19; 19; 32; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
        ↪  19; 19; 19; 19; 19; 32; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
         ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
         ↪  19; 19; 19; 19; 19; 19; 19; 33; 19; 19; 19;
         ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
         ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
         ↪  19; 19; 19; 19; 19; 19; 33; 19; 19; 19; 19;
         ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
         ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 34;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 34; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
     ↪  19; 19; 19; 35; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
     ↪  19; 19; 35; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
 ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
 ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
 ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
 ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
 ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
 ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
 ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
↪  19; 19; 19; 37; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
↪  19; 19; 37; 19; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 38; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  38; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  40; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 40;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 41; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 41; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 42; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 42; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
     [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
      ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
      ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
      ↪  19; 19; 19; 19; 19; 19; 19; 19; 43; 19; 19;
      ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
      ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
      ↪  19; 19; 19; 19; 19; 19; 19; 43; 19; 19; 19;
      ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 44; 19; 19; 19;
       ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 44; 19; 19; 19; 19;
       ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 45; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 45; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 46; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  46; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
     ↪  19; 19; 19; 47; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
     ↪  19; 19; 47; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 49;
    ↪  19; 50; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 51; 52; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 49; 19;
    ↪  50; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 51; 52; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 65;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 65; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 59; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 59; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 56; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 56; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 53; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 53; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; 19; 54; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; 54; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 55; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  55; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
      ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
      ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
      ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
      ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
      ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
      ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
      ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 57;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 57; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 58; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 58; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; 19; 60; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; 60; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 61; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 61; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
↪  19; 19; 19; 19; 19; 19; 19; 62; 19; 19; 19;
↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
↪  19; 19; 19; 19; 19; 19; 62; 19; 19; 19; 19;
↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 63; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 63; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
↪  19; 19; 19; 64; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
↪  19; 19; 64; 19; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 76; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 76; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 68; 19; 19; 19;
    ↪  19; 19; 69; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 68; 19; 19; 19; 19;
    ↪  19; 69; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
         ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
         ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
         ↪  19; 74; 19; 19; 19; 19; 19; 19; 19; 19; 19;
         ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
         ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
         ↪  74; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
         ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 70; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 70; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
      ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
      ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
      ↪  19; 19; 19; 19; 19; 71; 19; 19; 19; 19; 19;
      ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
      ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
      ↪  19; 19; 19; 19; 71; 19; 19; 19; 19; 19; 19;
      ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 72;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 72; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  73; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 73;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 75; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 75; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 77;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 77; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  78; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 78;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 79; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 79; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 80; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 80; 19; 19; 19; 19;
     ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
       ↪  19; 19; 19; 81; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
       ↪  19; 19; 81; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; 19; 82; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; 82; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 84; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 84; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 86; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 86; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 87; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 87; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
     ↪  19; 88; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
     ↪  88; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
 19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 89;
 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
 19; 19; 19; -1; -1; -1; -1; 19; -1; 89; 19;
 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 90;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 90; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
      ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
      ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
      ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
      ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
      ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
      ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
      ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 93; 19; 19; 19; 19; 94; 19; 19; 19;
        ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 93; 19; 19; 19; 19; 94; 19; 19; 19; 19;
        ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 101; 19;
    ↪  19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 101; 19;
    ↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
       ↪  19; 19; 19; 95; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
       ↪  19; 19; 95; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 96; 19; 19; 19; 19; 19;
    ↪  19; 97; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 96; 19; 19; 19; 19; 19; 19;
    ↪  97; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
↪  19; 19; 19; 99; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
↪  19; 19; 99; 19; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 98; 19; 19; 19;
        ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 98; 19; 19; 19; 19;
        ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪ -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
       ↪ 19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
       ↪ 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪ 19; 19; 19; 19; 19; 100; 19; 19; 19; 19;
       ↪ 19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
       ↪ 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪ 19; 19; 19; 19; 19; 100; 19; 19; 19; 19;
       ↪ 19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 102; 19; 19;
        ↪  19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 102; 19; 19;
        ↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
         19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
         19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
         19; 104; 19; 19; 19; 19; 19; 19; 19; 19;
         19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
         19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
         19; 104; 19; 19; 19; 19; 19; 19; 19; 19;
         19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪ -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
       ↪ 19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
       ↪ 19; 105; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪ 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪ 19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
       ↪ 19; 105; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪ 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪ 19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 106; 19; 19;
       ↪  19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 106; 19; 19;
       ↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
↪  19; 19; 19; 19; 19; 19; 19; 107; 19; 19;
↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
↪  19; 19; 19; 19; 19; 19; 19; 107; 19; 19;
↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 108; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 108; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 109; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 109; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 143; 19; 19; 19;
        ↪  19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 143; 19; 19; 19;
        ↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 112; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
    ↪  19; 19; 112; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  113; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; -1; -1; -1; -1; -1; -1;
       ↪  -1; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; -1; -1; -1; -1; 19; -1;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; 114; -1; 115; -1; -1; 116; -1; -1;
 ↪  -1; -1; -1; -1; 117; -1; -1; 118; -1; -1;
 ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; -1; 114; -1; 115; -1; -1; 116; -1;
 ↪  -1; -1; -1; -1; -1; 117; -1; -1; 118; -1;
 ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; 142; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; 142; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; 135; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; 135; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; 134; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; 134; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; 128; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; 128; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; 119; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; 119; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ 120; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ 120; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; 121; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; 121; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; 122; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; 122; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; 123; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; 123; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; 124; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; 124; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; 125; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; 125; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; 126; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; 126; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; 127; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; 127; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; 129; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; 129; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; 130; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; 130; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; 131; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; 131; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  132; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  132; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  133; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  133; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; 136; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; 136; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; 137; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; 137; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; 138; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; 138; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; 139; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; 139; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; 140; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; 140; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; 141; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; 141; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
↪  19; 19; 19; 144; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
↪  19; 19; 19; 144; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 146; 19;
        ↪  19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 146; 19;
        ↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  147; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
    ↪  147; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  148; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; 19; -1; -1; -1; -1; 19; -1;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  148; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 149; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
    ↪  19; 19; 19; 149; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ 150; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; 19; 19; 19; 19; 19;
↪ 19; 19; 19; 19; 19; -1; -1; -1; -1; -1; -1;
↪ -1; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪ 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪ 19; 19; 19; 19; 19; -1; -1; -1; -1; 19; -1;
↪ 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪ 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪ 19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 -1; -1; -1; 151; -1; -1; -1; -1; -1; -1;
 -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 -1; -1; -1; 151; -1; -1; -1; -1; -1; -1;
 -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; 152; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; 152; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; 153; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; 153; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; 154; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; 154; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; 155; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; 155; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; 156; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; 156; -1; -1; -1;
      ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; 157; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; 157; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; 158; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; 158; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; 159; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; 159; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪  181; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; 19; 19; -1; -1; -1; -1; 19; -1;
↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪  181; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪  19; 19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1;
    ↪  174; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; -1; -1; -1; -1; 19; -1;
    ↪  174; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  163; 164; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  163; 164; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 170; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
        ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 170; 19; 19; 19; 19; 19; 19;
        ↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 165; 19; 19;
       ↪  19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 165; 19; 19;
       ↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
↪ 19; 19; 19; -1; -1; -1; -1; -1; -1; -1;
↪ 166; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪ 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪ 19; 19; 19; 19; 19; -1; -1; -1; -1; 19; -1;
↪ 166; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪ 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪ 19; 19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 167; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 167; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
     19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
     19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     19; 168; 19; 19; 19; 19; 19; 19; 19; 19;
     19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
     19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     19; 168; 19; 19; 19; 19; 19; 19; 19; 19;
     19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
↪ 19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
↪ 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪ 19; 19; 19; 19; 19; 19; 169; 19; 19; 19;
↪ 19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
↪ 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
↪ 19; 19; 19; 19; 19; 19; 169; 19; 19; 19;
↪ 19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  171; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; -1; -1; -1; -1; 19; -1;
       ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  171; 19; 19; 19; 19; 19; 19; 19; 19; 19;
       ↪  19; 19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 172; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
    ↪  19; 19; 19; 172; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  173; 19; 19; -1; -1; -1; -1; 19; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  173; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪ 19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪ 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪ 19; 19; 19; 19; 19; 175; 19; 19; 19; 19;
    ↪ 19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
    ↪ 19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪ 19; 19; 19; 19; 19; 175; 19; 19; 19; 19;
    ↪ 19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
         ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1;
         ↪  176; 19; 19; 19; 19; 19; 19; 19; 19; 19;
         ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
         ↪  19; 19; 19; 19; 19; -1; -1; -1; -1; 19; -1;
         ↪  176; 19; 19; 19; 19; 19; 19; 19; 19; 19;
         ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
         ↪  19; 19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 177; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
    ↪  19; 177; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 178; 19; 19;
    ↪  19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
    ↪  19; 19; 19; 19; 19; 19; 19; 178; 19; 19;
    ↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
         ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
         ↪  19; 19; 19; 179; 19; 19; 19; 19; 19; 19;
         ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
         ↪  19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
         ↪  19; 19; 19; 179; 19; 19; 19; 19; 19; 19;
         ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
         ↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
 ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
 ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
 ↪  19; 19; 19; 19; 19; 180; 19; 19; 19; 19;
 ↪  19; 19; 19; 19; -1; -1; -1; -1; 19; -1; 19;
 ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
 ↪  19; 19; 19; 19; 19; 180; 19; 19; 19; 19;
 ↪  19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
         ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
         ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
         ↪  182; 19; 19; 19; 19; 19; 19; 19; 19; 19;
         ↪  19; 19; 19; 19; 19; -1; -1; -1; -1; 19; -1;
         ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
         ↪  182; 19; 19; 19; 19; 19; 19; 19; 19; 19;
         ↪  19; 19; 19; 19; 19; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; -1; -1; -1; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; -1; -1; -1; -1; 19; -1; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; 19; 19; 19; 19; 19; 19; 19; 19; 19;
     ↪  19; 19; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; 192; 192; 192; 192; 192;
        ↪  192; 192; 192; 192; 192; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; 193; 194; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ -1; -1; 186; -1; 187; 187; 187; 187; 187;
    ↪ 187; 187; 187; 187; 187; -1; -1; -1; -1;
    ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ -1; -1; -1; -1; -1; -1; 188; -1; -1; -1;
    ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  189; -1; 189; -1; -1; 190; 190; 190; 190;
    ↪  190; 190; 190; 190; 190; 190; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 191; 191; 191; 191; 191;
    ↪  191; 191; 191; 191; 191; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; 191; 191; 191; 191; 191;
        ↪  191; 191; 191; 191; 191; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; 191; 191; 191; 191; 191;
↪ 191; 191; 191; 191; 191; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; 192; 192; 192; 192; 192;
↪ 192; 192; 192; 192; 192; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; 193; 194; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  198; -1; 198; -1; -1; 199; 199; 199; 199;
 ↪  199; 199; 199; 199; 199; 199; -1; -1; -1;
 ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
 ↪  -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  195; -1; 195; -1; -1; 196; 196; 196; 196;
↪  196; 196; 196; 196; 196; 196; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; 197; 197; 197; 197; 197;
↪  197; 197; 197; 197; 197; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; 197; 197; 197; 197; 197;
↪  197; 197; 197; 197; 197; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; 197; 197; 197; 197; 197;
↪  197; 197; 197; 197; 197; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 200; 200; 200; 200; 200;
    ↪  200; 200; 200; 200; 200; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 200; 200; 200; 200; 200;
    ↪  200; 200; 200; 200; 200; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; 200; 200; 200; 200; 200;
↪  200; 200; 200; 200; 200; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 202; 202; 202; 202; 202;
    ↪  202; 202; 202; 202; 202; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; 193; 194; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; 238; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; 238; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; 235; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; 235; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  230; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  230; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; 227; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; 228; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; 227; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; 228;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; 224; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; 225; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; 224; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; 225;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; 217; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; 218; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 217; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; 218; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; 215; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; 215; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; 211; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; 211; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; 212; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; 212; -1;
     ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; 213; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; 213; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; 214; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; 216; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; 201; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; 221; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; 221; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; 219; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; 219; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; 220; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; 222;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; 222;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; 223; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; 185; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; 226; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; 183; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪ -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; 229; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  231; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  231; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; 232; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; 232; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; 233; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; 233; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; 234; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
    [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ -1; -1; 184; -1; -1; -1; -1; -1; -1; -1;
    ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ 236; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
    ↪ 236; -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; 237; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; 239; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; 239; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; 240; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
[|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
      [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; 5; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
       ↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; 4; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
         ↪  -1; -1; -1; -1; -1; -1; -1|];
```

```
        [|-1; -1; -1; -1; -1; -1; -1; -1; -1; -1; 1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  244; 3; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1; -1; -1; -1;
        ↪  -1; -1; -1; -1; -1; -1; -1; -1|]
    |]
}
```

```
open Environnement
open Abstract_tokens
open Bibliotheques
open Convert_to_abstract
open LL1
open Traduction

(** Renvoie les imports nécessaires contenus dans
↪  [l] *)
let rec generate_library_imports (l : libs) :
↪  string =
  match l with
  | [] -> ""
  | name :: q -> "#include <" ^ name ^ ".h>\n" ^
  ↪  generate_library_imports q

(** Renvoie le format de string dans les printf
↪  pour afficher la variable ou la
```

```
    fonction [n] à l'aide de [env] *)
let generate_format_string_of_name (n : string) (env : environnement) : string =
  match Hashtbl.find_opt env n with
  | Some (Syntax Real) -> "%f "
  | Some (Syntax Integer) -> "%d "
  | Some (Syntax Logical) -> "%d "
  | _ -> failwith "La variable n'est pas dans l'environnement"

(** Renvoie le format de string dans les printf pour afficher les éléments de
    [l] en s'aidant des types dans [env] *)
let rec generate_format_string (l : ast list) (env : environnement) : string =
  match l with
  | [] -> ""
```

```
    | Noeud (Chaine _, []) :: q -> "%s " ^ generate_format_string q env
    | Noeud (Integer _, []) :: q -> "%d " ^ generate_format_string q env
    | Noeud (Floating _, []) :: q -> "%f " ^ generate_format_string q env
    | Noeud (Booleen _, []) :: q -> "%d " ^ generate_format_string q env
    | Noeud (Name n, []) :: q ->
      generate_format_string_of_name n env ^ generate_format_string q env
    | Noeud (Syntax Call, Noeud (Name n, []) :: _) :: _ ->
      generate_format_string_of_name n env
    | Noeud (s, _) :: _ ->
      print_token s;
      failwith " type non def"
```

```ocaml

(** renvoie la chaîne associée au type de [var] dans [env] *)
let str_of_env_type (env : environnement) (var : string) : string =
  match Hashtbl.find_opt env var with
   | Some (Syntax Integer) -> "int"
   | Some (Syntax Real) -> "float"
   | Some t ->
       print_token t;
       failwith "type non supporté"
   | None -> failwith "var not found in environnement"

(** Applique la fonction [f] au dernier élément de la liste [l] *)
```

```
let rec map_to_last (f : 'a -> 'a) (l : 'a list) : 'a list =
  match l with
  | [] -> failwith "impossible to map to last"
  | e :: [] -> if e = L [] then failwith "e= []" else [ f e ]
  | e :: q -> e :: map_to_last f q

(** ajoute un point-virgule à la fin de [s] si elle n'en a pas déjà *)
let rec add_semi_colon (s : string_or_string_list) : string_or_string_list =
  match s with
  | S s -> if String.ends_with s ~suffix:";" then S s else S (s ^ ";")
  | L l -> L (map_to_last add_semi_colon l)

```

```
(** Renvoie le type de la fonction d'ast [a] à l'aide de l'environnement [env] *)
let get_function_return_type (a : ast) (env : environnement) :
    string_or_string_list =
  match a with
  | Noeud (Syntax Function, l) -> (
      let ret = last_of_list l in
      match ret with
      | Noeud (Syntax Return, [ Noeud (Name var_name, []) ]) ->
          S (str_of_env_type env var_name)
      | _ -> failwith "type de retour non pris en charge")
  | _ -> failwith "Cette fonction prend en paramètre un Noeud(Function)"
```

```
(** parcourt la liste des fils pour trouver les définitions des paramètres *)
let rec get_function_param_list (l : ast list) (env : environnement) :
    string_or_string_list * ast list =
  match l with

  | Noeud (Name nom, []) :: Noeud(Name n,[]) :: q ->
      let sosl, q2 = get_function_param_list (Noeud(Name n,[]) :: q) env in
      (L [ S (str_of_env_type env nom); S " "; S nom; S ", "; sosl ], q)

  | Noeud (Name nom, []) ::  q -> L [ S (str_of_env_type env nom); S " "; S nom ], q
  | l1 -> (S "", l)
```

```ocaml

(** convertit l'arbre de syntaxe abstrait [ast] à l'aide de l'environnement
    [env] et l'indente de [n_tab] en un string_or_string_list *)
let rec convert_ast_to_C_sosl (ast : ast list) (env : environnement)
    (nb_tab : int) : string_or_string_list =
  match ast with
  | [] -> S ""
  | [ Noeud (ProgramRoot, l) ] -> convert_ast_to_C_sosl l env nb_tab
  | Noeud (Syntax Program, Noeud (Name nom, []) :: l1) :: q ->
      L
        [
          Traduction.tabs_to_string nb_tab;
```

```
            S "// ";
            S nom;
            S "\n";
            Traduction.tabs_to_string nb_tab;
            S "void main(void){";
            convert_ast_to_C_sosl l1 env (nb_tab +
            ↪  1);
            S "}";
          ]
    | Noeud (Commentaire c, []) :: q ->
        L
          [
            Traduction.tabs_to_string nb_tab;
            S "//";
            S c;
            convert_ast_to_C_sosl q env nb_tab;
          ]
```

```
    | Noeud (Syntax Print, l2) :: q ->
        L
          [
            Traduction.tabs_to_string nb_tab;
            S "printf(\"";
            S (generate_format_string l2 env);
            S "\"";
            L
              (List.rev
                 (List.fold_left
                    (fun acc x ->
                      convert_ast_to_C_sosl [ x ] env nb_tab :: S ", " :: acc)
                    [] l2));
            S ");";
            convert_ast_to_C_sosl q env nb_tab;
          ]
```

```
(* définit le type des variables *)
| Noeud (Syntax Real, l) :: q ->
   let s = convert_ast_to_C_sosl l env 0 in
   let s = add_semi_colon s in
   L
     [
       Traduction.tabs_to_string nb_tab;
       S "float ";
       s;
       convert_ast_to_C_sosl q env nb_tab;
     ]
| Noeud (Syntax Integer, l) :: q ->
   let s = convert_ast_to_C_sosl l env 0 in
   let s = add_semi_colon s in
   L
     [
       Traduction.tabs_to_string nb_tab;
```

```
                S "int ";
                s;
                convert_ast_to_C_sosl q env nb_tab;
              ]
   | Noeud (Syntax Logical, l) :: q ->
        let s = convert_ast_to_C_sosl l env 0 in
        let s = add_semi_colon s in
        L
          [
            Traduction.tabs_to_string nb_tab;
            S "bool ";
            s;
            convert_ast_to_C_sosl q env nb_tab;
          ]
   | Noeud (Syntax Double_precision, l) :: q ->
        let s = convert_ast_to_C_sosl l env 0 in
        let s = add_semi_colon s in
```

```
        L
          [
            Traduction.tabs_to_string nb_tab;
            S "long ";
            s;
            convert_ast_to_C_sosl q env nb_tab;
          ]
    | Noeud (Syntax Character, Noeud (Syntax Constant, []) :: l) :: q ->
        let s = convert_ast_to_C_sosl l env 0 in
        let s = add_semi_colon s in
        L
          [
            Traduction.tabs_to_string nb_tab;
            S "const char ";
            s;
            convert_ast_to_C_sosl q env nb_tab;
```

```
        ]
    | Noeud (Syntax Character, l) :: q ->
        let s = convert_ast_to_C_sosl l env 0 in
        let s = add_semi_colon s in
        L
          [
            Traduction.tabs_to_string nb_tab;
            S "char ";
            s;
            convert_ast_to_C_sosl q env nb_tab;
          ]
    | Noeud (Operateur Assignation, Noeud (Name s, []) :: l) :: q ->
        L
          [
            Traduction.tabs_to_string nb_tab;
            S s;
```

```
      S " = ";
      convert_ast_to_C_sosl l env 0;
      S ";";
      convert_ast_to_C_sosl q env nb_tab;
     ]
  | Noeud (Syntax Size, l) :: q ->
    L
     [
      Traduction.tabs_to_string nb_tab;
      S "[";
      convert_ast_to_C_sosl l env 0;
      S "] ";
      convert_ast_to_C_sosl q env nb_tab;
     ]
  | Noeud (Operateur Assignation, Noeud (Syntax
  ↪  Size, l1) :: l) :: q ->
    L
```

```
            [
              Traduction.tabs_to_string nb_tab;
              convert_ast_to_C_sosl [ Noeud (Syntax
               ↪  Size, l1) ] env 0;
              convert_ast_to_C_sosl
                (Noeud (Operateur Assignation, l) :: q)
                env nb_tab;
            ]
    | Noeud (Syntax Any, []) :: q ->
     ↪  convert_ast_to_C_sosl q env 0
    | Noeud (Name s, []) :: q -> L [ S s;
     ↪  convert_ast_to_C_sosl q env nb_tab ]
    | Noeud (Operateur Plus, elem :: l) :: q ->
        L
            [
              convert_ast_to_C_sosl [ elem ] env
               ↪  nb_tab;
```

```
              S " + ";
              convert_ast_to_C_sosl l env nb_tab;
              convert_ast_to_C_sosl q env nb_tab;
            ]

  (* opérateur unaire *)
  | Noeud (Operateur Moins, elem :: []) :: q ->
      L
        [
          S " -";
          convert_ast_to_C_sosl [ elem ] env
           ↪ nb_tab;
          convert_ast_to_C_sosl q env nb_tab;
        ]

  | Noeud (Operateur Moins, elem :: l) :: q ->
      L
```

```
          [
            convert_ast_to_C_sosl [ elem ] env
            ↪ nb_tab;
            S " - ";
            convert_ast_to_C_sosl l env nb_tab;
            convert_ast_to_C_sosl q env nb_tab;
          ]
    | Noeud (Operateur Fois, elem :: l) :: q ->
        L
          [
            convert_ast_to_C_sosl [ elem ] env
            ↪ nb_tab;
            S " * ";
            convert_ast_to_C_sosl l env nb_tab;
            convert_ast_to_C_sosl q env nb_tab;
          ]
    | Noeud (Operateur Division, elem :: l) :: q ->
```

```
        L
          [
            convert_ast_to_C_sosl [ elem ] env nb_tab;
            S " / ";
            convert_ast_to_C_sosl l env nb_tab;
            convert_ast_to_C_sosl q env nb_tab;
          ]
    | Noeud (Parentheseouvrante, []) :: q ->
        L [ S "("; convert_ast_to_C_sosl q env nb_tab ]
    | Noeud (Parenthesefermante, []) :: q ->
        L [ S ")"; convert_ast_to_C_sosl q env nb_tab ]
    | Noeud (OperateurLogique NonEquivalent, [ p1; p2 ]) :: q
    | Noeud (Comparateur NonEgal, [ p1; p2 ]) :: q ->
```

```
        L
          [
            convert_ast_to_C_sosl [ p1 ] env 0;
            S " != ";
            convert_ast_to_C_sosl [ p2 ] env 0;
            convert_ast_to_C_sosl q env nb_tab;
          ]
    | Noeud (OperateurLogique Equivalent, [ p1; p2 ])
    ↪  :: q
    | Noeud (Comparateur Egal, [ p1; p2 ]) :: q ->
        L
          [
            convert_ast_to_C_sosl [ p1 ] env 0;
            S " == ";
            convert_ast_to_C_sosl [ p2 ] env 0;
            convert_ast_to_C_sosl q env nb_tab;
          ]
```

```
  | Noeud (Comparateur StrictPlusPetit, [ p1; p2 ]) :: q ->
      L
        [
          convert_ast_to_C_sosl [ p1 ] env 0;
          S " < ";
          convert_ast_to_C_sosl [ p2 ] env 0;
          convert_ast_to_C_sosl q env nb_tab;
        ]
  | Noeud (Comparateur PlusPetit, [ p1; p2 ]) :: q ->
      L
        [
          convert_ast_to_C_sosl [ p1 ] env 0;
          S " <= ";
          convert_ast_to_C_sosl [ p2 ] env 0;
          convert_ast_to_C_sosl q env nb_tab;
```

```
            ]
    | Noeud (Comparateur StrictPlusGrand, [ p1; p2 ])
    ↪   :: q ->
        L
          [
            convert_ast_to_C_sosl [ p1 ] env 0;
            S " > ";
            convert_ast_to_C_sosl [ p2 ] env 0;
            convert_ast_to_C_sosl q env nb_tab;
          ]
    | Noeud (Comparateur PlusGrand, [ p1; p2 ]) :: q
    ↪   ->
        L
          [
            convert_ast_to_C_sosl [ p1 ] env 0;
            S " >= ";
            convert_ast_to_C_sosl [ p2 ] env 0;
```

```
                convert_ast_to_C_sosl q env nb_tab;
              ]
    | Noeud (OperateurLogique Et, [ p1; p2 ]) :: q ->
        L
          [
            convert_ast_to_C_sosl [ p1 ] env 0;
            S " && ";
            convert_ast_to_C_sosl [ p2 ] env 0;
            convert_ast_to_C_sosl q env nb_tab;
          ]
    | Noeud (OperateurLogique Ou, [ p1; p2 ]) :: q ->
        L
          [
            convert_ast_to_C_sosl [ p1 ] env 0;
            S " || ";
            convert_ast_to_C_sosl [ p2 ] env 0;
            convert_ast_to_C_sosl q env nb_tab;
```

```
          ]
    | Noeud (OperateurLogique Non, [ p1; p2 ]) :: q
    ↪   ->
        L
          [
            convert_ast_to_C_sosl [ p1 ] env 0;
            S " ! ";
            convert_ast_to_C_sosl [ p2 ] env 0;
            convert_ast_to_C_sosl q env nb_tab;
          ]
    | Noeud (Operateur Puissance, [ p1; p2 ]) :: q ->
        L
          [
            S "pow((long)";
            convert_ast_to_C_sosl [ p1 ] env 0;
            S ", ";
            convert_ast_to_C_sosl [ p2 ] env 0;
```

```
        S "(long))";
        convert_ast_to_C_sosl q env nb_tab;
       ]
    | Noeud (Syntax If, condition :: instructions) :: q ->
      L
       [
        Traduction.tabs_to_string nb_tab;
        S "if (";
        convert_ast_to_C_sosl [ condition ] env nb_tab;
        S "){";
        convert_ast_to_C_sosl instructions env (nb_tab + 1);
        Traduction.tabs_to_string nb_tab;
        S "}";
        convert_ast_to_C_sosl q env nb_tab;
```

```
            ]
    | Noeud (Syntax Else_if, condition ::
    ↪  instructions) :: q ->
        L
            [
              Traduction.tabs_to_string nb_tab;
              S "else if (";
              convert_ast_to_C_sosl [ condition ] env
              ↪  nb_tab;
              S "){";
              convert_ast_to_C_sosl instructions env
              ↪  (nb_tab + 1);
              Traduction.tabs_to_string nb_tab;
              S "}";
              convert_ast_to_C_sosl q env nb_tab;
            ]
    | Noeud (Syntax Else, instructions) :: q ->
```

```
            L
              [
                Traduction.tabs_to_string nb_tab;
                S "else {";
                convert_ast_to_C_sosl instructions env
                 ↪  (nb_tab + 1);
                Traduction.tabs_to_string nb_tab;
                S "}";
                convert_ast_to_C_sosl q env nb_tab;
              ]
      | Noeud
          ( Syntax For,
            Noeud (Operateur Assignation, [ variable;
             ↪  valeur ])
            :: fin
            :: Noeud (Syntax Step, [ pas ])
            :: instructions )
```

```
      :: q ->
        L
          [
            Traduction.tabs_to_string nb_tab;
            S "for (";
            convert_ast_to_C_sosl
              [ Noeud (Operateur Assignation, [ variable; valeur ]) ]
              env 0;
            S " ";
            convert_ast_to_C_sosl
              [ Noeud (Comparateur StrictPlusPetit, [ variable; fin ]) ]
              env 0;
            S "; ";
            convert_ast_to_C_sosl [ variable ] env 0;
            S "=";
```

```
          convert_ast_to_C_sosl
            [ Noeud (Operateur Plus, [ variable;
            ↪  pas ]) ]
            env 0;
          S ") {";
          convert_ast_to_C_sosl instructions env
          ↪  (nb_tab + 1);
          Traduction.tabs_to_string nb_tab;
          S "}";
          convert_ast_to_C_sosl q env nb_tab;
        ]
  | Noeud (Syntax While, condition :: instructions)
  ↪  :: q ->
      L
        [
          Traduction.tabs_to_string nb_tab;
          S "while (";
```

```
        convert_ast_to_C_sosl [ condition ] env 0;
        S "){";
        convert_ast_to_C_sosl instructions env (nb_tab + 1);
        Traduction.tabs_to_string nb_tab;
        S "}";
        convert_ast_to_C_sosl q env nb_tab;
      ]
| Noeud (Integer s, []) :: q -> L [ S s; convert_ast_to_C_sosl q env nb_tab ]
| Noeud (Floating s, []) :: q -> L [ S s; convert_ast_to_C_sosl q env nb_tab ]
| Noeud (Double s, []) :: q ->
    let d =
      String.fold_left
```

```
          (fun acc x -> if x = 'd' then acc ^ "e" else acc ^ String.make 1 x)
          "" s
      in
      L [ S d; convert_ast_to_C_sosl q env nb_tab ]
      (* convertit les d en e de fortran *)
  | Noeud (Booleen b, []) :: q ->
      L
        [
          S (if b then "true" else "false"); convert_ast_to_C_sosl q env nb_tab;
        ]
  | Noeud (Chaine s, []) :: q -> L [ S s; convert_ast_to_C_sosl q env nb_tab ]
  | Noeud (Syntax Function, Noeud (Name n, []) :: l) :: q ->
```

```
        let sosl, l2 = get_function_param_list l env
        ↪  in
        L
          [
            tabs_to_string nb_tab;
            get_function_return_type
              (Noeud (Syntax Function, Noeud (Name n,
              ↪  []) :: l))
              env;
            S " ";
            S n;
            S "(";
            sosl;
            S "){\n";
            convert_ast_to_C_sosl l2 env (nb_tab +
            ↪  1);
            tabs_to_string nb_tab;
```

```
                S "}\n";
              ]
    | Noeud (Syntax Call, Noeud (Name n, []) :: l) ::
    ↪  q ->
        L
          [
            S n;
            S "(";
            convert_ast_to_C_sosl l env 0;
            S ")";
            convert_ast_to_C_sosl q env nb_tab;
          ]
    | Noeud (Syntax Return, l) :: q ->
        L
          [
            tabs_to_string nb_tab;
            S "return ";
```

```
          convert_ast_to_C_sosl l env 0;
          S ";\n";
          convert_ast_to_C_sosl q env nb_tab;
        ]

  | Noeud (NewLine, []) :: q -> L [ S "\n"; convert_ast_to_C_sosl q env nb_tab ]
  | Noeud (t, _) :: q ->
      print_token t;
      failwith "La syntaxe donnée n'est pas encore prise en charge\n"

(** convertit l'arbre de syntaxe abstrait [ast] et ajoute les librairies
    nécessaires depuis [biblios] grace à [env] *)
let convert_ast_to_C (ast : ast list) (env : environnement)
```

```
    (biblios : Bibliotheques.libs) : string =
  generate_library_imports biblios
  ^ Traduction.string_of_string_or_string_list
  ↪  (convert_ast_to_C_sosl ast env 0)


```

```
open Environnement
open Abstract_tokens
open Bibliotheques
open Convert_to_abstract
open LL1
open Traduction

let rec convert_ast_to_Fortran_sosl (ast : ast list) (env : environnement)
    (nb_tab : int) : string_or_string_list =
  match ast with
  | [] -> S ""
  | [ Noeud (ProgramRoot, l) ] -> convert_ast_to_Fortran_sosl l env nb_tab
  | Noeud (Syntax Program, Noeud (Name nom, []) :: l1) :: q ->
      L
```

```
            [
              S "program ";
              S nom;
              convert_ast_to_Fortran_sosl l1 env
              ↪  (nb_tab + 1);
              S "end program\n";
              convert_ast_to_Fortran_sosl q env nb_tab;
            ]
    | Noeud (Commentaire c, []) :: q ->
        L
          [
            Traduction.tabs_to_string nb_tab;
            S "!";
            S c;
            convert_ast_to_Fortran_sosl q env nb_tab;
          ]
    | Noeud (Syntax Print, l2) :: q ->
```

```
        L
          [
            Traduction.tabs_to_string nb_tab;
            S "print *";
            L
              (List.rev
                (List.fold_left
                  (fun acc x ->
                    convert_ast_to_Fortran_sosl [ x ] env nb_tab
                    :: S ", " :: acc)
                  [] l2));
            convert_ast_to_Fortran_sosl q env nb_tab;
          ]
    (* définit le type des variables *)
    | Noeud (Syntax Real, l) :: q ->
        let s = convert_ast_to_Fortran_sosl l env 0 in
```

```
        L
          [
            Traduction.tabs_to_string nb_tab;
            S "real ::";
            s;
            convert_ast_to_Fortran_sosl q env nb_tab;
          ]
    | Noeud (Syntax Integer, l) :: q ->
        let s = convert_ast_to_Fortran_sosl l env 0
          ↪ in
        L
          [
            Traduction.tabs_to_string nb_tab;
            S "integer ::";
            s;
            convert_ast_to_Fortran_sosl q env nb_tab;
          ]
```

```
    | Noeud (Syntax Logical, l) :: q ->
        let s = convert_ast_to_Fortran_sosl l env 0
         ↪  in
        L
          [
            Traduction.tabs_to_string nb_tab;
            S "logical ::";
            s;
            convert_ast_to_Fortran_sosl q env nb_tab;
          ]
    | Noeud (Syntax Double_precision, l) :: q ->
        let s = convert_ast_to_Fortran_sosl l env 0
         ↪  in
        L
          [
            Traduction.tabs_to_string nb_tab;
            S "double precision ::";
```

```
                s;
                convert_ast_to_Fortran_sosl q env nb_tab;
            ]
  | Noeud (Operateur Assignation, Noeud (Name s, []) :: l) :: q ->
      L
        [
          Traduction.tabs_to_string nb_tab;
          S s;
          S " = ";
          convert_ast_to_Fortran_sosl l env 0;
          convert_ast_to_Fortran_sosl q env nb_tab;
        ]
  | Noeud (Name s, []) :: q ->
      L [ S s; convert_ast_to_Fortran_sosl q env nb_tab ]
  | Noeud (Operateur Plus, elem :: l) :: q ->
```

```
    L
      [
        convert_ast_to_Fortran_sosl [ elem ] env nb_tab;
        S " + ";
        convert_ast_to_Fortran_sosl l env nb_tab;
        convert_ast_to_Fortran_sosl q env nb_tab;
      ]
| Noeud (Operateur Moins, elem :: l) :: q ->
    L
      [
        convert_ast_to_Fortran_sosl [ elem ] env nb_tab;
        S " - ";
        convert_ast_to_Fortran_sosl l env nb_tab;
        convert_ast_to_Fortran_sosl q env nb_tab;
      ]
```

```
  | Noeud (Operateur Fois, elem :: l) :: q ->
    L
      [
        convert_ast_to_Fortran_sosl [ elem ] env
        ↪  nb_tab;
        S " * ";
        convert_ast_to_Fortran_sosl l env nb_tab;
        convert_ast_to_Fortran_sosl q env nb_tab;
      ]
  | Noeud (Operateur Division, elem :: l) :: q ->
    L
      [
        convert_ast_to_Fortran_sosl [ elem ] env
        ↪  nb_tab;
        S " / ";
        convert_ast_to_Fortran_sosl l env nb_tab;
        convert_ast_to_Fortran_sosl q env nb_tab;
```

```
        ]
    | Noeud (Parentheseouvrante, []) :: q ->
      L [ S "("; convert_ast_to_Fortran_sosl q env nb_tab ]
    | Noeud (Parenthesefermante, []) :: q ->
      L [ S ")"; convert_ast_to_Fortran_sosl q env nb_tab ]
    | Noeud (OperateurLogique NonEquivalent, [ p1; p2 ]) :: q ->
      L
        [
          convert_ast_to_Fortran_sosl [ p1 ] env 0;
          S " .neqv. ";
          convert_ast_to_Fortran_sosl [ p2 ] env 0;
          convert_ast_to_Fortran_sosl q env nb_tab;
        ]
    | Noeud (Comparateur NonEgal, [ p1; p2 ]) :: q ->
```

```
        L
          [
            convert_ast_to_Fortran_sosl [ p1 ] env 0;
            S " /= ";
            convert_ast_to_Fortran_sosl [ p2 ] env 0;
            convert_ast_to_Fortran_sosl q env nb_tab;
          ]
    | Noeud (OperateurLogique Equivalent, [ p1; p2 ])
    ↪   :: q ->
        L
          [
            convert_ast_to_Fortran_sosl [ p1 ] env 0;
            S " .eqv. ";
            convert_ast_to_Fortran_sosl [ p2 ] env 0;
            convert_ast_to_Fortran_sosl q env nb_tab;
          ]
    | Noeud (Comparateur Egal, [ p1; p2 ]) :: q ->
```

```
        L
          [
            convert_ast_to_Fortran_sosl [ p1 ] env 0;
            S " == ";
            convert_ast_to_Fortran_sosl [ p2 ] env 0;
            convert_ast_to_Fortran_sosl q env nb_tab;
          ]
    | Noeud (Comparateur StrictPlusPetit, [ p1; p2 ]) :: q ->
        L
          [
            convert_ast_to_Fortran_sosl [ p1 ] env 0;
            S " < ";
            convert_ast_to_Fortran_sosl [ p2 ] env 0;
            convert_ast_to_Fortran_sosl q env nb_tab;
          ]
    | Noeud (Comparateur PlusPetit, [ p1; p2 ]) :: q ->
```

```
      L
        [
          convert_ast_to_Fortran_sosl [ p1 ] env 0;
          S " <= ";
          convert_ast_to_Fortran_sosl [ p2 ] env 0;
          convert_ast_to_Fortran_sosl q env nb_tab;
        ]
  | Noeud (Comparateur StrictPlusGrand, [ p1; p2 ])
  ↪  :: q ->
      L
        [
          convert_ast_to_Fortran_sosl [ p1 ] env 0;
          S " > ";
          convert_ast_to_Fortran_sosl [ p2 ] env 0;
          convert_ast_to_Fortran_sosl q env nb_tab;
        ]
  | Noeud (Comparateur PlusGrand, [ p1; p2 ]) :: q
  ↪  ->
```

```
        L
          [
            convert_ast_to_Fortran_sosl [ p1 ] env 0;
            S " >= ";
            convert_ast_to_Fortran_sosl [ p2 ] env 0;
            convert_ast_to_Fortran_sosl q env nb_tab;
          ]
    | Noeud (OperateurLogique Et, [ p1; p2 ]) :: q ->
        L
          [
            convert_ast_to_Fortran_sosl [ p1 ] env 0;
            S " && ";
            convert_ast_to_Fortran_sosl [ p2 ] env 0;
            convert_ast_to_Fortran_sosl q env nb_tab;
          ]
    | Noeud (OperateurLogique Ou, [ p1; p2 ]) :: q ->
        L
```

```
            [
              convert_ast_to_Fortran_sosl [ p1 ] env 0;
              S " || ";
              convert_ast_to_Fortran_sosl [ p2 ] env 0;
              convert_ast_to_Fortran_sosl q env nb_tab;
            ]
    | Noeud (OperateurLogique Non, [ p1; p2 ]) :: q
    ↪   ->
          L
            [
              convert_ast_to_Fortran_sosl [ p1 ] env 0;
              S " ! ";
              convert_ast_to_Fortran_sosl [ p2 ] env 0;
              convert_ast_to_Fortran_sosl q env nb_tab;
            ]
    | Noeud (Operateur Puissance, [ p1; p2 ]) :: q ->
          L
```

```
          [
            convert_ast_to_Fortran_sosl [ p1 ] env 0;
            S "**";
            convert_ast_to_Fortran_sosl [ p2 ] env 0;
            convert_ast_to_Fortran_sosl q env nb_tab;
          ]
    | Noeud (Syntax If, condition :: instructions)
      :: Noeud (Syntax Else_if, l)
      :: q ->
        L
          [
            Traduction.tabs_to_string nb_tab;
            S "if (";
            convert_ast_to_Fortran_sosl [ condition ] env nb_tab;
            S ") do";
            convert_ast_to_Fortran_sosl instructions env (nb_tab + 1);
```

```
                Traduction.tabs_to_string nb_tab;
                convert_ast_to_Fortran_sosl
                  (Noeud (Syntax Else_if, l) :: q)
                  env nb_tab;
            ]
    | Noeud (Syntax If, condition :: instructions) ::
    ↪  Noeud (Syntax Else, l) :: q
      ->
        L
          [
            Traduction.tabs_to_string nb_tab;
            S "if (";
            convert_ast_to_Fortran_sosl [ condition ]
            ↪  env nb_tab;
            S ") do";
            convert_ast_to_Fortran_sosl instructions
            ↪  env (nb_tab + 1);
```

```
        Traduction.tabs_to_string nb_tab;
        convert_ast_to_Fortran_sosl (Noeud (Syntax Else, l) :: q) env nb_tab;
      ]
| Noeud (Syntax If, condition :: instructions) :: q ->
    L
      [
        Traduction.tabs_to_string nb_tab;
        S "if (";
        convert_ast_to_Fortran_sosl [ condition ] env nb_tab;
        S ") do";
        convert_ast_to_Fortran_sosl instructions env (nb_tab + 1);
        Traduction.tabs_to_string nb_tab;
        S "end if";
```

```
              convert_ast_to_Fortran_sosl q env nb_tab;
            ]
    | Noeud (Syntax Else_if, condition :: instructions)
      :: Noeud (Syntax Else, l)
      :: q ->
        L
          [
              Traduction.tabs_to_string nb_tab;
              S "else if (";
              convert_ast_to_Fortran_sosl [ condition ] env nb_tab;
              S ")";
              convert_ast_to_Fortran_sosl instructions env (nb_tab + 1);
              Traduction.tabs_to_string nb_tab;
```

```
            convert_ast_to_Fortran_sosl (Noeud
             ↪  (Syntax Else, l) :: q) env nb_tab;
          ]
   | Noeud (Syntax Else_if, condition ::
   ↪  instructions)
     :: Noeud (Syntax Else_if, l)
     :: q ->
       L
         [
            Traduction.tabs_to_string nb_tab;
            S "else if (";
            convert_ast_to_Fortran_sosl [ condition ]
             ↪  env nb_tab;
            S ")";
            convert_ast_to_Fortran_sosl instructions
             ↪  env (nb_tab + 1);
            Traduction.tabs_to_string nb_tab;
```

```
        convert_ast_to_Fortran_sosl
          (Noeud (Syntax Else_if, l) :: q)
          env nb_tab;
      ]
| Noeud (Syntax Else_if, condition :: instructions) :: q ->
    L
      [
        Traduction.tabs_to_string nb_tab;
        S "else if (";
        convert_ast_to_Fortran_sosl [ condition ] env nb_tab;
        S ")";
        convert_ast_to_Fortran_sosl instructions env (nb_tab + 1);
        Traduction.tabs_to_string nb_tab;
        S "end if";
```

```
            convert_ast_to_Fortran_sosl q env nb_tab;
          ]
  | Noeud (Syntax Else, instructions) :: q ->
      L
        [
            Traduction.tabs_to_string nb_tab;
            S "else ";
            convert_ast_to_Fortran_sosl instructions
            ↪  env (nb_tab + 1);
            Traduction.tabs_to_string nb_tab;
            S "end if";
            convert_ast_to_Fortran_sosl q env nb_tab;
          ]
  | Noeud
      ( Syntax For,
        Noeud (Operateur Assignation, [ variable;
        ↪  valeur ])
```

```
          :: fin
          :: Noeud (Syntax Step, [ pas ])
          :: instructions )
      :: q ->
        L
          [
            Traduction.tabs_to_string nb_tab;
            S "do ";
            convert_ast_to_Fortran_sosl
              [ Noeud (Operateur Assignation, [ variable; valeur ]) ]
              env 0;
            S ", ";
            convert_ast_to_Fortran_sosl [ fin ] env 0;
            S ", ";
            convert_ast_to_Fortran_sosl [ pas ] env 0;
```

```
            S "\n";
            convert_ast_to_Fortran_sosl instructions env (nb_tab + 1);
            Traduction.tabs_to_string nb_tab;
            S "end do";
            convert_ast_to_Fortran_sosl q env nb_tab;
          ]
    | Noeud (Syntax While, condition :: instructions) :: q ->
        L
          [
            Traduction.tabs_to_string nb_tab;
            S "do while (";
            convert_ast_to_Fortran_sosl [ condition ] env 0;
            S ")";
```

```
          convert_ast_to_Fortran_sosl instructions env (nb_tab + 1);
          Traduction.tabs_to_string nb_tab;
          S "end do";
          convert_ast_to_Fortran_sosl q env nb_tab;
        ]
  | Noeud (Integer s, []) :: q ->
      L [ S s; convert_ast_to_Fortran_sosl q env nb_tab ]
  | Noeud (Floating s, []) :: q ->
      L [ S s; convert_ast_to_Fortran_sosl q env nb_tab ]
  | Noeud (Double s, []) :: q ->
      L [ S s; convert_ast_to_Fortran_sosl q env nb_tab ]
      (* convertit les d en e de fortran *)
  | Noeud (Booleen b, []) :: q ->
```

```
      L
        [
          S (if b then "true" else "false");
          convert_ast_to_Fortran_sosl q env nb_tab;
        ]
  | Noeud (Chaine s, []) :: q ->
      L [ S s; convert_ast_to_Fortran_sosl q env nb_tab ]
  | Noeud (NewLine, []) :: q ->
      L [ S "\n"; convert_ast_to_Fortran_sosl q env nb_tab ]
  | _ -> failwith "La syntaxe donnée n'est pas encore prise en charge\n"

let convert_ast_to_Fortran (ast : ast list) (env : environnement) : string =
  Traduction.string_of_string_or_string_list
```

```
    (convert_ast_to_Fortran_sosl ast env 0)
```

```ocaml
open TraductionFortran
open TraductionC
open Det_automaton
open LL1
open Convert_to_abstract
open Environnement
open Automates
open Grammar

let transpile_Fortran_to_C (fortran_file_name :
↪  string) (c_file_name : string) :
    unit =
  let lexemes = exec_of_file syntax_automate_det
  ↪  fortran_file_name in
  let arbre_syntaxique = analyse_LL1 grammar
  ↪  lexemes in
  let arbre_syntaxique_abstrait =
  ↪  convert_to_abstract arbre_syntaxique in
```

```
  let env = create_env_from_ast
   ↪  arbre_syntaxique_abstrait in
  (* print_env env;  *)
  let code_C =
    convert_ast_to_C
      [ arbre_syntaxique_abstrait ]
      (env)
      []
  in
  let out_file = open_out c_file_name in
  output_string out_file code_C;
  close_out out_file

let transpile_Fortran_to_Fortran (fortran_file_name
 ↪   : string)
    (c_file_name : string) : unit =
  let lexemes = exec_of_file syntax_automate_det
   ↪  fortran_file_name in
```

```
  let arbre_syntaxique = analyse_LL1 grammar lexemes in
  let arbre_syntaxique_abstrait = convert_to_abstract arbre_syntaxique in
  let env = create_env_from_ast arbre_syntaxique_abstrait in

  let code_Fortran =
    convert_ast_to_Fortran
      [ arbre_syntaxique_abstrait ]
      (env)
  in
  let out_file = open_out c_file_name in
  output_string out_file code_Fortran;
  close_out out_file
```

```
open Transpileurs

let print_usage () =
  print_string "Usage : ";
  print_newline ();
  print_string Sys.argv.(0);
  print_string " -C <inupt_fortran_file> [-o <output_c_file>]";
  print_newline ();
  print_string Sys.argv.(0);
  print_string " -Fortran <inupt_fortran_file> [-o <output_fortran_file>]";
  print_newline ()

let output_file (len : int) : string =
  if len == 5 then
    if
```

```
      (* un nom de fichier de sortie est donné *)
      Sys.argv.(3) = "-o"
    then Sys.argv.(4)
    else (
      print_usage ();
      raise (Invalid_argument ""))
  else
    (* un nom de fichier de sortie n'est pas donné, on en donne un par défaut *)
    match Sys.argv.(1) with
    | "-Fortran" -> "out.f90"
    | "-C" -> "out.c"
    | _ ->
        print_usage ();
        raise (Invalid_argument "")

let main () =
```

```
  let len = Array.length Sys.argv in
  if len <> 3 && len <> 5 then print_usage ()
  else
    let input_file_name = Sys.argv.(2) in
    let output_file_name = output_file len in
    print_string ("Generating " ^ output_file_name ^ "...");
    print_newline ();
    if Sys.argv.(1) = "-C" then
      transpile_Fortran_to_C input_file_name output_file_name
    else transpile_Fortran_to_Fortran input_file_name output_file_name;
    print_string ("Finished generating " ^ output_file_name);
    print_newline ()

```

```
let _ = main ()
```