

DAUPS

Documentation

Table des matières

Chapitre 1 : Introduction	3
Chapitre 2 : Syntaxe	5
Chapitre 3 : Fonctions intégrées	10
Chapitre 4 : Erreurs	14
Chapitre 5 : Exemples	18
Chapitre 6 : Exécution	36

Chapitre 1 : Introduction

Introduction au langage DAUPS

DAUPS est un langage de programmation pédagogique conçu pour favoriser l'apprentissage structuré de l'algorithmique. Il permet d'écrire, tester et exécuter des algorithmes à l'aide d'une syntaxe rigide et typée, proche des standards universitaires.

Développé dans un cadre académique, DAUPS s'adresse principalement aux étudiants en informatique de niveau débutant, afin de leur offrir une transition naturelle entre la théorie algorithmique et la programmation effective.

Objectifs du langage

DAUPS a été conçu pour :

- Encourager une rédaction rigoureuse des algorithmes,
- Imposer la déclaration explicite des variables avec un **typage statique**,
- Faciliter la compréhension des erreurs via des messages clairs à l'exécution,
- Offrir un environnement de développement complet via une **extension Visual Studio Code dédiée**.

Caractéristiques principales

- **Typage statique explicite** : chaque variable doit être déclarée avec un type (`int`, `float`, `string`, etc.) avant utilisation.
- **Déclarations obligatoires** : toute variable non déclarée ou mal utilisée entraîne une erreur à l'exécution.
- **Structure stricte** : les programmes commencent par `Algo`, contiennent un bloc `Begin ... End`, et peuvent inclure des fonctions typées.
- **Détection d'erreurs** : l'interpréteur vérifie la cohérence du typage, la portée des variables et la validité des appels de fonction.
- **Écosystème intégré** : l'exécution est assurée par un interpréteur Python, et l'extension VS Code fournit un environnement interactif avec coloration syntaxique, autocomplétion, info-bulles et exécution directe.

Exécution d'un programme DAUPS

Pour exécuter du code DAUPS :

1. Installez l'extension depuis le [marketplace Visual Studio Code](#).
2. Créez un nouveau fichier avec l'extension `.daups`.
3. Rédigez votre algorithme en respectant la syntaxe du langage.
4. Cliquez sur le bouton ▶ dans la barre supérieure pour lancer l'interpréteur.

L'environnement signale automatiquement les erreurs de syntaxe, de typage ou d'exécution, et propose des suggestions contextuelles via l'autocomplétion et les infobulles.

Ressources complémentaires

- **Interpréteur DAUPS** : moteur d'exécution écrit en Python.
- **Extension VS Code** : améliore la lisibilité et l'interaction avec le code.
- **Documentation en ligne** : régulièrement mise à jour et disponible dans plusieurs langues.

Si vous avez une question, une suggestion ou constatez une erreur dans cette documentation, vous pouvez [créer une issue sur GitHub](#).

Chapitre 2 : Syntaxe

Syntaxe du langage DAUPS

Le langage DAUPS est un langage de pseudocode à structure rigide. Il est conçu pour être facilement lisible et proche du langage naturel, tout en imposant une structure stricte pour garantir une exécution correcte.

Structure générale

Un programme commence par le mot-clé `Algo`, suivi éventuellement de déclarations de variables. Le bloc principal est délimité par les mots-clés `Begin` et `End`.

```
Algo
  Déclaration_de_variables
Begin
  Instructions
End
```

Commentaires

Les commentaires commencent par `#` et s'étendent jusqu'à la fin de la ligne.

```
# Ceci est un commentaire
get x # Ceci aussi
```

Types de base

Les types de données suivants sont pris en charge :

- `int` : entier
- `float` : réel
- `bool` : booléen
- `string` : chaîne de caractères
- `array of T` : tableau de type `T`, où `T` est l'un des types précédents (y compris un tableau)

```
x : int
texte : string
liste : array of int
matrice : array of array of float
```

Déclaration de variables

Les variables doivent être déclarées **avant le bloc Begin, dans le bloc Algo**.
Plusieurs variables du même type peuvent être déclarées ensemble, séparées par des virgules.

```
Algo
  identificateur_1 : type_1
  identificateur_2_1, identificateur_2_2 : type_2
Begin
  Instructions
End
```

Affectation

L'opérateur d'affectation est <--.

```
Algo
  identificateur_1 : type_1
  identificateur_2_1, identificateur_2_2 : type_2
Begin
  identificateur_1 <-- valeur
  identificateur_2_1 <-- expression
End
```

Entrées / Sorties

Lecture (get)

```
get identificateur
get tableau[indexe]
```

Affichage (print)

```
print "Valeur de identificateur :", identificateur, "Saut-de-ligne"
```

"Saut-de-ligne" est une chaîne spéciale pour un retour à la ligne.

Contrôle de flux

Conditionnelle (if, else if, else)

```
if Condition then
    Instructions
else if Condition then
    Instructions
else
    Instructions
```

Exemple :

```
if x > 0 then
    print "Positif"
if x == 0 then
    print "Nul"
else
    print "Négatif"
```

Boucles

Tant que (while)

```
while (Condition)
    Instructions
```

Pour (for)

```
for i <-- 0 to 10
    print i
```

```
for i <-- 10 downto 0
  print i
```

Fonctions

Une fonction est définie par le mot-clé `function`, suivie du nom, des paramètres typés, du type de retour (s'il y en a un), et d'un bloc `Begin / End`.

```
function nom(param_1 : type_1, ...) : type_retour
  var : type
  Begin
    Instructions
  return valeur
End
```

Exemple :

```
function maximum(a : int, b : int) : int
  Begin
    if a > b then
      return a
    else
      return b
  End
```

Tableaux

Déclaration

```
t : array of int
mat : array of array of int
```

Création

```
t <-- create_array(5)
mat <-- create_array(3, 4)
```


Accès

```
t[0] <-- 42  
get mat[i][j]
```

Opérations autorisées par type

Type	Opérations possibles
int	+, -, *, /, div, mod, ==, <, >...
float	+, -, *, /, ==, <, >...
bool	and, or, not, comparaisons
string	+ (concaténation), comparaisons (==, !=, etc.)
array	accès par indice t[i], taille size(t), création

Remarques importantes

- La casse est **sensible** (Variable, variable, VARIABLE ne désignent pas la même chose).
- L'indentation est **obligatoire**
- Le mot-clé `End` clôt tous les blocs (programme principal et fonctions).
- Les fonctions peuvent s'appeler entre elles ou être utilisées dans le bloc principal.
- Une erreur est levée si une variable est utilisée sans avoir été déclarée.

Chapitre 3 : Fonctions intégrées

Fonctions intégrées du langage DAUPS

Le langage **DAUPS** propose plusieurs **fonctions intégrées** (*builtins*), accessibles directement sans redéfinition. Elles sont automatiquement chargées dans la table des symboles globale à chaque exécution.

Liste des fonctions intégrées

Nom	Description	Nombre d'arguments	Exemple d'appel
print	Affiche un ou plusieurs éléments sans retour à la ligne	0 ou plus (illimité)	print "Salut", x
get	Lit une valeur utilisateur et l'affecte à une variable	1 (obligatoire)	get x OU get tab[i][j]
create_array	Crée un tableau vide de taille donnée	≥1 (illimité)	tab <-- create_array(3, 4)
run	Exécute un autre fichier .daups (ou .txt)	1 (chemin string)	run "exemple.daups"
SQRT	Calcule la racine carrée d'un nombre	1 (numérique)	print SQRT(25)
nombreAleatoire	Génère un entier aléatoire entre deux bornes	2 (numériques)	nombreAleatoire(1, 100)
size	Retourne la taille d'un tableau ou d'une dimension spécifique	1 ou 2 (tableau, [dim])	size(tab) OU size(tab, 1)
Pi	Constante mathématique (π)	0 (aucun)	print Pi

Détails des fonctions

◆ print ...

Affiche des valeurs sans retour automatique à la ligne.

- **Arguments** : 0 ou plus (string, int, float, etc.)

"Saut-de-ligne" est une chaîne spéciale pour un retour à la ligne.

- **Exemple** :

```
print "Bonjour", nom, 42
```

◆ get x, get tab[i]

Demande une entrée utilisateur, typée dynamiquement selon la variable ciblée.

- **Arguments** : 1 (variable cible)
- **Exemple** :

```
get age  
get matrice[i][j]
```

◆ create_array(dim1, dim2, ...)

Crée un tableau vide à une ou plusieurs dimensions.

- **Arguments** : ≥ 1 (chaque argument représente une dimension, de type int)
- **Exemples** :

```
tab <- create_array(5)           # tableau 1D  
mat <- create_array(4, 3)        # tableau 2D  
cube <- create_array(2, 2, 2)    # tableau 3D
```

◆ run "chemin"

Exécute un fichier DAUPS externe.

- **Arguments** : 1 (string - chemin vers un fichier .daups ou .txt)
- **Exemple** :

```
run "mon_fichier.daups"
```

◆ SQRT(valeur)

Renvoie la racine carrée d'un nombre.

- **Arguments** : 1 (int ou float)
- **Exemple** :

```
r <-- Sqrt(16)
```

◆ **nombreAleatoire(min, max)**

Retourne un entier aléatoire compris dans l'intervalle [min, max].

- **Arguments** : 2 (int ou float)
- **Exemple** :

```
n <-- nombreAleatoire(1, 10)
```

◆ **size(tableau[, dimension])**

Renvoie la taille totale d'un tableau ou d'une dimension spécifique.

- **Arguments** : 1 (tableau) ou plus (tableau, dimension_1, ...)
- **Exemples** :

```
print size(tab)           # taille totale  
print size(tab, 1)        # taille de la 1ère dimension
```

◆ **Pi**

Constante flottante équivalente à $\pi \approx 3.141592653589793$.

- **Arguments** : 0
- **Exemple** :

```
print Pi
```

Chapitre 4 : Erreurs

Gestion des erreurs dans le langage DAUPS

L'interpréteur DAUPS est conçu pour détecter et signaler les erreurs fréquentes à l'exécution. Grâce à une analyse statique et dynamique, il fournit des messages explicites facilitant le débogage et la compréhension du comportement du code.

Types d'erreurs détectées

Type d'erreur	Description
Variable non déclarée	Utilisation d'une variable absente des déclarations
Variable non initialisée	Lecture d'une variable avant affectation
Conflit de type	Affectation ou opération incompatible avec le type attendu
Appel de fonction inconnu	Appel d'une fonction non définie ou mal orthographiée
Mauvais nombre d'arguments	Appel de fonction avec trop ou pas assez de paramètres
Accès hors tableau	Tentative d'accès à un indice inexistant
Type non pris en charge	Usage d'un type inexistant ou mal formé
Expression invalide	Syntaxe incorrecte dans une affectation ou un test conditionnel
Erreur utilisateur (entrée)	Saisie d'une valeur non conforme au type de la variable
Mauvaise instruction	Mot-clé ou structure non reconnue

Exemples d'erreurs et messages correspondants

Variable non déclarée

```
Algo
Begin
  x <-- 5  # x n'a pas été déclaré
End
```

RunTime error: Variable 'x' is not declared

```
x <-- 5  # x n'a pas été déclaré
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Conflit de type

```
Algo
  x : int
Begin
  x <-- "texte"
End
```

RunTime error: Variable 'x' is of type 'int', but got 'String'

```
x <-- "texte"
^^^^^^
```

Appel de fonction inconnu

```
Algo
  resultat : int
Begin
  resultat <-- inconnu(3)
End
```

RunTime error: 'inconnu' is not defined

```
resultat <-- inconnu(3)
^^^^^^^^^^
```

Mauvais nombre d'arguments

```
function f(a : int, b : int) : int
  Begin
    return a + b
  End

Algo
Begin
  print f(3)  # il manque un argument
End
```

RunTime error: 1 too few arguments passed into 'f'
Expected 2 arguments, got 1

```
print f(3)  # il manque un argument
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Accès hors tableau

```
Algo
  tab : array of int
Begin
  tab <-- create_array(3)
  print tab[5]  # hors bornes
End
```

RunTime error: Index access error (probably out of bounds)

```
print tab[5]  # hors bornes
^^^^^^^^^^^^
```

Déclenchement des erreurs

Les erreurs sont déclenchées soit :

- **Lors de l'analyse statique** (déclarations, types)
- **À l'exécution** (accès mémoire, appels dynamiques, saisie utilisateur)

L'interpréteur interrompt l'exécution dès qu'une erreur est détectée, tout en indiquant précisément **la ligne concernée**, **le type d'erreur**, ainsi que **la variable ou la fonction impliquée**.

Bonnes pratiques pour éviter les erreurs

- Déclarer systématiquement les variables avec leur type avant `Begin`.
- Respecter les types lors des affectations et des appels de fonctions.
- Vérifier la taille des tableaux avant tout accès par indice.
- Lire attentivement les messages d'erreur, conçus pour être explicites.

Chapitre 5 : Exemples

Exemples DAUPS

Exemple 1

```
Algo
  x : int
Begin
  print "Donner une valeur entiere"
  get x
  x <-- x+1
  print x
End
```

Exemple 2

```
Algo
  x, y : float
Begin
  print "donnez une valeur entiere"
  get x
  y <-- 3*x+1
  print y
End
```

Exemple 3

```
Algo
  x, y, temp : float
Begin
  print "Donner des valeurs numeriques"
  get x
  get y
  temp <-- x
  x <-- y
  y <-- temp
End
```

Exemple 4

```
Algo
  a, b, c, D, x1, x2 : float
Begin
  print "Quel est le parametre a ?"
  get a
  print "Quel est le parametre b ?"
  get b
  print "Quel est le parametre c ?"
  get c
  D <-- (b*b - 4*a*c)
  if (D < 0) then
    print "Delta est negatif et l'equation n'admet aucune racine reelle"
  else if D == 0 then
    print "Delta = 0 et l'equation admet une solution double x = ", -b/(2*a)
  else
    x1 <-- (-b - SQRT(D)) / (2*a)
    x2 <-- (-b + SQRT(D)) / (2*a)
    print "Delta positif, l'equation admet 2 solutions reelles et distinctes",
End
```

Exemple 5

```
Algo
  a, b, c : int
Begin
  print "Saisir un entier"
  get a
  print "Saisir un entier"
  get b
  print "Saisir un entier"
  get c
  if (a==(b+c) or b==(a+c) or c==(a+b)) then
    print "oui"
  else
    print "non"
End
```

Exemple 6

Algo

a, b, c, d, e, f, x, y : int

Begin

get a

get b

get c

get d

get e

get f

if ((d*b)-(e*a))==0 then

Les droites ont la même pente

Les coeff directeurs sont égaux

$a/b == d/e \Leftrightarrow ae = db$

if b==0 and e==0 then

cas de 2 droites verticales

if (c*d)==(f*a) then

2 droites verticales confondues

print "Infinite de solutions"

else

les 2 droites verticales sont parallèles

print "pas de solution"

else

if ((b*f)-(e*c))==0 then

droite confondues

print "Infinite de solutions"

else

les droites sont parallèles

print "Pas de solutions"

else

$((db-ea) \neq 0)$

x <- ((b*f)-(e*c))/((d*b)-(e*a))

if b == 0 then

$e \neq 0$

y <- (f/e)-(d/e)*x

else

y <- (c/b)-(a/b)*x

print "x=", x

print "y=", y

End

Exemple 7

```
Algo
  n : float
Begin
  print "Saisir un entier"
  get n
  if (n<10) then
    print "Ajourné"
  else
    if n < 12 then
      print "Passable"
    else
      if n<14 then
        print "AB"
      else
        if n<16 then
          print "B"
        else
          print "TB"
End
```

Exemple 8

```
Algo
#calcule  $x^n$ 
  x : float
  n : int
  i : int #compteur
  r : float #variable stockant le résultat
Begin
  print "Quelle est la valeur de x ?"
  get x
  while (x<0)
    print "x ne peut pas être négatif, entrez une autre valeur !"
    get x
  print "Quelle est la valeur de n ?"
  get n
  while (n<0)
    print "n ne peut pas être négatif, entrez une autre valeur!"
    get n
  i <-- 0
  r <-- 1
  while (i<n)
    r <-- r*x
    i <-- i+1
  print r
End
```

Exemple 9

```
Algo
# somme des impaires < 100
  cpt, r : int
Begin
  cpt <-- 1
  r <-- 0
  while (cpt<= 100)
    if (cpt mod 2) != 0 then
      r <-- r + cpt
    cpt <-- cpt+1
  print r
End
```

Exemple 10

```
Algo
# Est-ce un nombre premier ?
n, d, somme : int
Begin
  get n
  d <-- 1
  somme <-- 0
  while (d < n)
    if (n mod d) == 0 then
      somme <-- somme + d
    d <-- d+1
  print "La somme des diviseurs est :"
  print somme
  if somme == 1 then
    print "Ce nombre est premier."
End
```

Exemple 11

```
Algo
# Tous les nombres parfaits  $\leq n$ 
n, d, somme : int
Begin
  get n
  while (n > 1)
    # Chercher les diviseurs de ce nombre n
    d <-- 1
    somme <-- 0
    while (d < n)
      if ((n mod d) == 0) then
        # d est un diviseur de n
        somme <-- somme + d
      d <-- d + 1
    if (somme == n) then
      # n est un nombre parfait
      print n
    n <-- n - 1
End
```

Exemple 12

```
Algo
# Tous les multiples de 7 entre i et j
  i, j : int
Begin
  print "Debut : "
  get i
  print "Fin : "
  get j
  while (i < j)
    if ((i mod 7) == 0) then
      print i
      print " est un multiple de 7."
      print "Saut-de-ligne"
    i <-- i+1
End
```

Exemple 13

```
Algo
# Affichage du tableau
  nbLignes : int # numéro de la ligne courante
  i : int # entier à afficher
  nbCol : int # numéro de la colonne courante
Begin
  nbLignes <-- 1
  while (nbLignes <= 4)
    nbCol <-- 1
    i <-- 1
    while (nbCol <= 5)
      print i
      print " "
      i <-- i + nbLignes
      nbCol <-- nbCol + 1
    print "Saut-de-ligne"
    nbLignes <-- nbLignes + 1
End
```


Exemple 14

```
Algo
# Suite croissante ?
n : int
i : int # compteur
p : int # nombre précédemment saisi
c : int # nombre courant saisi
b : bool # vrai tant que la suite est triée
Begin
  n <-- 0
  while (n <= 0)
    get n
  get c
  i <-- 1 # On a déjà saisi un entier - reste (n-1)
  b <-- True
  while (i < n)
    p <-- c # on stocke l'entier précédemment saisi
    get c
    if (p > c) then
      # si l'entier est < au précédent
      b <-- False # Valeurs non-ordonnées de manière croissante.
    i <-- i + 1
  if (b == True) then
    print "Les", n, "valeurs sont triées de façon croissante."
  else
    print "Les", n, "valeurs ne sont pas triées."
End
```

Exemple 15

Algo

Somme des pairs == Somme des impairs ?

n : int

si : int # sommes des entiers impairs

sp : int # sommes des entiers pairs

i : int # compteur

e : int # entier saisi

Begin

n <-- 0

si <-- 0

sp <-- 0

while (n <= 0)

print "Saisir une valeur positive non nulle :"

get n

i <-- 0

while (i < n)

e <-- 0

while (e <= 0)

print "Saisir un entier positif non nul :"

get e

if (e mod 2 == 0) then

sp <-- sp + e

else

si <-- si + e

i <-- i + 1

if (si == sp) then

print "La somme des nombres pairs est égale à la somme des nombres impairs."

else

print "La somme des nombres pairs n'est pas égale à la somme des nombres impairs."

End

Exemple 16

```
function maximum(n1 : float, n2 : float) : float
  Begin
    if (n1 > n2) then
      return n1
    else
      return n2
    End
  End

Algo
  n1, n2 : float
  Begin
    print "Saisir deux valeurs"
    get n1
    get n2
    print maximum(n1, n2)
  End
```

Exemple 17

```
function cube(x : float) : float
  Begin
    return x ** 3
  End

function volume(r : float) : float
  Begin
    return (4 / 3) * Pi * cube(r)
  End

Algo
  r : float
  Begin
    print "Saisir le rayon"
    get r
    print volume(r)
  End
```

Exemple 18

```
function tableMulti(base : int, debut : int, fin : int)
  n : float
  Begin
    print "Fragment de la table de multiplication par", base, ": "
    n <-- debut
    while (n <= fin)
      print n, "x", base, "=", n * base
      print "Saut-de-ligne"
      n <-- n + 1
    End
  End

Algo
  b, d, f : int
  Begin
    print "Saisir la base, le début et la fin"
    get b
    get d
    get f
    tableMulti(b, d, f)
  End
```

Exemple 19

```
function pgcdParDiviseurs(a : int, b : int) : int
  pgcd : int
  Begin
    pgcd <-- b
    while (pgcd > 1)
      if (a mod pgcd == 0 and b mod pgcd == 0) then
        return pgcd
      pgcd <-- pgcd - 1
    return 1
  End

function maximum(a : int, b : int) : int
  Begin
    if (a > b) then
      return a
    else
      return b
  End

function minimum(a : int, b : int) : int
  Begin
    if (a < b) then
      return a
    else
      return b
  End

function pgcdParDifferences(a : int, b : int) : int
  diff : int
  Begin
    diff <-- a - b
    while (diff > 0)
      a <-- maximum(diff, b)
      b <-- minimum(diff, b)
      diff <-- a - b
    return a
  End

function pgcdParEuclide(a : int, b : int) : int
  reste : int
  Begin
    reste <-- a mod b
    while (reste > 0)
      a <-- b
      b <-- reste
      reste <-- a mod b
    return b
```

```
End

Algo
  a, b : int
Begin
  get a
  get b
  print pgcdParDiviseurs(a, b), "Saut-de-ligne"
  print pgcdParDifferences(a, b), "Saut-de-ligne"
  print pgcdParEuclide(a, b)
End
```

Exemple 20

Algo

```
taille, i, dessus, indice : int
somme, moyenne, grand : float
note : array of float
```

Begin

```
get taille
note <-- create_array(taille)
somme <-- 0

for i <-- 0 to taille - 1
    print "Note en position ", i+1, " ?"
    get note[i]
    somme <-- somme + note[i]

moyenne <-- somme / taille
print "la moyenne est", moyenne, "Saut-de-ligne"

dessus <-- 0
for i <-- 0 to taille - 1
    if note[i] >= moyenne then
        dessus <-- dessus + 1
print "le nombre de notes au-dessus de la moyenne est", dessus, "Saut-de-ligne"

grand <-- note[0]
indice <-- 0
for i <-- 1 to taille - 1
    if note[i] > grand then
        grand <-- note[i]
        indice <-- i

print "le nombre maximum est", grand, "Saut-de-ligne"
print "il est en position", indice + 1, "Saut-de-ligne"
```

End

Exemple 21

```
Algo
  i, long : int
  tab : array of int
Begin
  get long
  tab <-- create_array(long)

  for i <-- 0 to long - 1
    get tab[i]
    if tab[i] >= 0 then
      print tab[i], "Saut-de-ligne"
End
```

Exemple 22

```
Algo
  i, long, long2 : int
  tab, tab2 : array of int
Begin
  get long
  tab <-- create_array(long)
  tab2 <-- create_array(long)
  long2 <-- 0

  for i <-- 0 to long - 1
    print "Élément en position ", i + 1, " ?", "Saut-de-ligne"
    get tab[i]
    if tab[i] >= 0 then
      tab2[long2] <-- tab[i]
      long2 <-- long2 + 1

  print "le nouveau tableau contient ", long2, " éléments positifs", "Saut-de-ligne"
  tab <-- tab2

  for i <-- 0 to long2 - 1
    print tab[i], "Saut-de-ligne"
End
```


Exemple 23

Algo

```
i, j, long : int  
tab : array of int
```

Begin

```
get long  
tab <-- create_array(long)  
for i <-- 0 to long - 1  
    print "Élément en position ", i + 1, " ?"  
    get tab[i]
```

```
j <-- 0  
for i <-- 0 to long - 1  
    if tab[i] >= 0 then  
        tab[j] <-- tab[i]  
        j <-- j + 1
```

```
long <-- j  
for i <-- 0 to long - 1  
    print tab[i], "Saut-de-ligne"
```

End

Exemple 24

```
function tabAlea(n : int, a : int, b : int) : array of int
  T : array of int
  i : int
Begin
  T <-- create_array(n)
  for i <-- 0 to n - 1
    T[i] <-- nombreAleatoire(a, b)
  return T
End

function tabProduit(T : array of int) : int
  produit, i : int
Begin
  produit <-- 1
  for i <-- 0 to size(T) - 1
    produit <-- produit * T[i]
  return produit
End

Algo
  a, b, n, i, produit : int
  T : array of int
Begin
  print "Saisir les trois valeurs", "Saut-de-ligne"
  get n
  get a
  get b
  T <-- tabAlea(n, a, b)
  produit <-- tabProduit(T)
  for i <-- 0 to n - 1
    print T[i], "Saut-de-ligne"
  print produit, "Saut-de-ligne"
End
```

Exemple 25

```
Algo
  i, j : int
  tab : array of int
Begin
  tab <-- create_array(4, 2)
  for i <-- 0 to 3
    for j <-- 0 to 1
      tab[i][j] <-- 2 * i + j
  for i <-- 0 to 3
    for j <-- 0 to 1
      print tab[i][j], "Saut-de-ligne"
End
```

Exemple 26

```
Algo
  i, j, grand : int
  tab : array of int
Begin
  tab <-- create_array(12, 8)
  for i <-- 0 to 11
    for j <-- 0 to 7
      print "Quel est l'élément de la ligne ", i + 1, " et de la colonne ", j + 1, ": ",
        get tab[i][j]

  grand <-- tab[0][0]
  for i <-- 0 to 11
    for j <-- 0 to 7
      if tab[i][j] > grand then
        grand <-- tab[i][j]
  print "le nombre maximum est ", grand, "Saut-de-ligne"
End
```

Chapitre 6 : Exécution

Déroulement de l'exécution dans le langage DAUPS

L'interpréteur DAUPS exécute un programme en respectant une structure rigide et typée, dans laquelle les blocs, les variables, les fonctions et les instructions sont validés à la fois statiquement et dynamiquement.

Ce chapitre décrit le pipeline complet d'exécution, depuis le chargement d'un fichier `.daups` jusqu'à la production des résultats.

Étapes principales de l'exécution

1. Chargement du fichier
2. Parsing / Tokenisation
3. Analyse syntaxique
4. Analyse sémantique (types, fonctions, portées...)
5. Construction de la table des symboles
6. Exécution ligne par ligne
7. Gestion des erreurs et affichage des résultats

1. Chargement du fichier source

Les fichiers DAUPS sont généralement des fichiers `.daups` ou `.txt` contenant une structure de ce type :

```
Algo
  a, b : int
Begin
  get a
  get b
  print a + b
End
```

Le fichier est lu en entier, puis les lignes sont traitées séquentiellement.

2. Analyse syntaxique et construction du bloc

L'interpréteur détecte les blocs principaux (Algo, Begin, End) et construit une structure logique du programme, incluant :

- Les instructions simples (get, print, `x <-- 3`)
- Les blocs de contrôle (if, while, for)
- Les définitions de fonctions
- Les déclarations de variables

3. Table des symboles

Les variables déclarées et fonctions définies sont enregistrées dans une table des symboles :

- Globalement pour l'ensemble du programme
- Par portée locale pour chaque fonction

Chaque symbole est associé à :

- un nom
- un type (int, float, etc.)
- une valeur (initiale ou déterminée à l'exécution)
- un contexte (local ou global)

4. Exécution des instructions

Le bloc principal est exécuté dans l'ordre d'écriture. Chaque ligne peut correspondre à :

- une affectation
- une lecture utilisateur
- une instruction de contrôle
- un appel de fonction (intégrée ou utilisateur)

Les expressions sont évaluées dynamiquement avec contrôle de type.

5. Appel de fonctions

Lorsqu'une fonction est appelée :

- Les arguments sont évalués
- Un contexte local est créé
- Les variables locales sont isolées de l'environnement global
- Une valeur est retournée, si la fonction en prévoit une

Exemple :

```
function f(x : int) : int
  Begin
    return x * 2
  End

Algo
  y : int
  Begin
    y <-- f(3)
    print y
  End
```

6. Gestion des erreurs

Toute erreur détectée pendant l'exécution (variable non déclarée, type incompatible, indice hors limites, etc.) interrompt immédiatement le programme, avec un message d'erreur explicite.

7. Fin de l'exécution

Lorsque toutes les instructions du bloc principal sont exécutées :

- Les résultats sont affichés dans la console (ou capturés si redirigés)
- Le programme se termine normalement si aucune erreur n'a été levée

Exemple d'un programme complet exécuté

```
Algo
  a, b : int
  resultat : int
Begin
  get a
  get b
  resultat <-- a + b
  print "Résultat :", resultat
End
```

Résumé

L'exécution dans DAUPS suit une structure stricte mais prévisible :

- Pas de compilation : tout est interprété dynamiquement
- Les types et les portées sont strictement respectés
- Le contrôle des erreurs est systématique
- Les fonctions et tableaux sont gérés dynamiquement

Fait par PerseusShade

Licence MIT

Copyright (c) 2025 PerseusShade

La présente autorisation est accordée, gratuitement, à toute personne obtenant une copie de ce logiciel et des fichiers de documentation associés (le « Logiciel »), de traiter le Logiciel sans restriction, y compris sans limitation les droits d'utiliser, copier, modifier, fusionner, publier, distribuer, sous-licencier et/ou vendre des copies du Logiciel, et de permettre aux personnes à qui le Logiciel est fourni de le faire, sous réserve des conditions suivantes :

La notice de droit d'auteur ci-dessus et la présente notice d'autorisation doivent être incluses dans toutes copies ou parties substantielles du Logiciel.

LE LOGICIEL EST FOURNI « EN L'ÉTAT », SANS GARANTIE D'AUCUNE SORTE, EXPLICITE OU IMPLICITE, Y COMPRIS MAIS SANS S'Y LIMITER LES GARANTIES DE QUALITÉ MARCHANDE, D'ADÉQUATION À UN USAGE PARTICULIER ET D'ABSENCE DE CONTREFAÇON. EN AUCUN CAS LES AUTEURS OU LES TITULAIRES DES DROITS D'AUTEUR NE POURRONT ÊTRE TENUS RESPONSABLES DE TOUTE RÉCLAMATION, DOMMAGE OU AUTRE RESPONSABILITÉ, QU'IL S'AGISSE D'UNE ACTION CONTRACTUELLE, DÉLICTUELLE OU AUTRE, DÉCOULANT DE, OU EN LIEN AVEC LE LOGICIEL OU L'UTILISATION OU AUTRES MANIÈRES DE TRAITER DANS LE LOGICIEL.

Note : traduction non officielle

Cette version française n'est donnée qu'à titre informatif. Il n'existe pas de version officielle en français de la licence MIT ; seule la version en anglais fait foi et prévaut en cas de litige.