DAUPS

Documentation

Made by PerseusShade

Table of Contents

Chapter 1 : Introduction

Introduction to the DAUPS Language

DAUPS is an educational programming language designed to promote structured learning of algorithmic thinking. It allows users to write, test, and execute algorithms using a strict, statically-typed syntax aligned with academic standards.

Developed in an academic context, DAUPS is primarily intended for beginner-level computer science students, providing a natural bridge between theoretical algorithms and practical programming.

Language Objectives

DAUPS was designed to:

- Encourage rigorous writing of algorithms,
- Enforce explicit variable declarations with **static typing**,
- Improve error comprehension through clear runtime messages,
- Provide a complete development environment via a **dedicated Visual Studio Code extension**.

Key Features

- **Explicit static typing**: each variable must be declared with a type (`int`, `float`, `string`, etc.) before use.
- **Mandatory declarations**: any undeclared or improperly used variable results in a runtime error.
- **Strict structure**: programs must start with `Algo`, contain a `Begin ... End` block, and may include typed functions.
- **Error detection**: the interpreter verifies type consistency, variable scope, and the validity of function calls.
- **Integrated ecosystem**: execution is handled by a Python-based interpreter, and the VS Code extension provides an interactive environment with syntax highlighting, autocompletion, tooltips, and direct execution.

Running a DAUPS Program

To run DAUPS code:

1. Install the official extension from the Visual Studio Code marketplace: https://vscode.com/extension.
2. Create a new file with the `.daups` extension.
3. Write your algorithm following the language's syntax.
4. Click the ▷ button in the top bar to launch the interpreter.

The environment automatically reports syntax, type, or runtime errors and offers contextual suggestions through autocompletion and tooltips.

Additional Resources

- **DAUPS Interpreter**: a Python-based execution engine.
- **VS Code Extension**: enhances code readability and interaction.
- **Online Documentation**: regularly updated and available in multiple languages.

If you have a question, suggestion, or notice an issue in this documentation, you may open an issue here: https://github.com/yourusername/daups-extension/issues

Chapter 2 : Syntax

DAUPS Language Syntax

The DAUPS language is a pseudocode language with a rigid structure. It is designed to be easily readable and close to natural language, while enforcing a strict structure to guarantee correct execution.

General Structure

A program begins with the keyword `Algo`, optionally followed by variable declarations. The main block is delimited by the keywords `Begin` and `End`.

```
Algo
    Variable_declarations
Begin
    Instructions
End
```

Comments

Comments start with `#` and extend to the end of the line.

```
# This is a comment
get x  # This is also a comment
```

Basic Types

The following data types are supported:

- `int`: integer
- `float`: real number
- `bool`: boolean
- `string`: string of characters
- `array of T`: array of type `T`, where `T` is one of the aforementioned types (including another array)

```
x : int
text : string
list : array of int
matrix : array of array of float
```

Variable Declaration

Variables must be declared **before the `Begin` block, within the `Algo` block**. Multiple variables of the same type can be declared together, separated by commas.

```
Algo
    identifier_1 : type_1
    identifier_2_1, identifier_2_2 : type_2
Begin
    Instructions
End
```

Assignment

The assignment operator is `<--`.

```
Algo
    identifier_1 : type_1
    identifier_2_1, identifier_2_2 : type_2
Begin
    identifier_1 <-- value
    identifier_2_1 <-- expression
End
```

Input / Output

Reading (`get`)

```
get identifier
get array[index]
```

Printing (`print`)

```
print "Identifier value:", identifier, "Saut-de-ligne"
```

`"Saut-de-ligne"` is a special string representing a line break.

Control Flow

Conditional (`if`, `else if`, `else`)

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

Example:

```
if x > 0 then
    print "Positive"
if x == 0 then
    print "Zero"
else
    print "Negative"
```

Loops

While Loop (`while`)

```
while (Condition)
    Instructions
```

For Loop (`for`)

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

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

Functions

A function is defined by the keyword `function`, followed by its name, typed parameters, return type (if any), and a `Begin`/`End` block.

```
function name(param_1 : type_1, ...) : return_type
        var : type
    Begin
        Instructions
        return value
    End
```

Example:

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

Arrays

Declaration

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

Creation

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

Access

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

Operations Permitted by Type

Type	Permitted Operations
`int`	`+`, `-`, `*`, `/`, `div`, `mod`, `==`, `<`, `>...`
`float`	`+`, `-`, `*`, `/`, `==`, `<`, `>...`
`bool`	`and`, `or`, `not`, comparisons
`string`	`+` (concatenation), comparisons (`==`, `!=`, etc.)
`array`	index access `t[i]`, size `size(t)`, creation

Important Notes

- Case is **sensitive** (`Variable`, `variable`, `VARIABLE` are distinct).
- Indentation is **mandatory**.
- The keyword `End` closes all blocks (main program and functions).
- Functions may call one another or be used within the main block.
- An error is raised if a variable is used without prior declaration.

Chapter 3 : Built in functions

Built-in Functions of the DAUPS Language

The **DAUPS** language provides several **built-in** functions, accessible directly without redefinition. They are automatically loaded into the global symbol table at each execution.

List of Built-in Functions

Name	Description	Number of Arguments	Example Call
`print`	Displays one or more items without automatically adding a newline	0 or more (unlimited)	`print "Hello", x`
`get`	Reads a user input and assigns it to a variable	1 (mandatory)	`get x` or `get tab[i][j]`
`create_array`	Creates an empty array of a given size	≥1 (unlimited)	`tab <-- create_array(3, 4)`
`run`	Executes another `.dps` (or `.txt`) file	1 (string path)	`run "example.dps"`
`SQRT`	Computes the square root of a number	1 (numeric)	`print SQRT(25)`
`nombreAleatoire`	Generates a random integer between two bounds	2 (numeric)	`nombreAleatoire(1, 100)`
`size`	Returns the size of an array or a specific dimension	1 or 2 (array, [dim])	`size(tab)` or `size(tab, 1)`
`Pi`	Mathematical constant (π)	0 (none)	`print Pi`

Details of Functions

- **`print ...`**

Displays values without automatically appending a newline.

- **Arguments**: 0 or more (`string`, `int`, `float`, etc.)

    `"Saut-de-ligne"` is a special string representing a newline.

- **Example**:

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

`get x, get tab[i]`

Prompts the user for input, dynamically typed according to the targeted variable.

- **Arguments**: 1 (target variable)
- **Example**:

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

`create_array(dim1, dim2, ...)`

Creates an empty array with one or more dimensions.

- **Arguments**: ≥1 (each argument represents a dimension, of type `int`)
- **Examples**:

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

`run "path"`

Executes an external DAUPS file.

- **Arguments**: 1 (`string` – path to a `.dps` or `.txt` file)
- **Example**:

```
run "my_file.dps"
```

`SQRT(value)`

Returns the square root of a number.

- **Arguments**: 1 (`int` or `float`)
- **Example**:

```
r <-- SQRT(16)
```

◆ `nombreAleatoire(min, max)`

Returns a random integer in the interval `[min, max]`.

- **Arguments**: 2 (`int` or `float`)
- **Example**:

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

◆ `size(array[, dimension])`

Returns the total size of an array or the size of a specific dimension.

- **Arguments**: 1 (array) or 2 (array, dimension)
- **Examples**:

```
print size(tab)          # total size
print size(tab, 1)       # size of the 1st dimension
```

◆ `Pi`

Floating-point constant equivalent to `π ≈ 3.141592653589793`.

- **Arguments**: 0
- **Example**:

```
print Pi
```

Chapter 4 : Errors

Error Handling in the DAUPS Language

The DAUPS interpreter is designed to detect and report common runtime errors. Through static and dynamic analysis, it provides explicit messages that facilitate debugging and understanding of code behavior.

Types of Detected Errors

Error Type	Description
Undeclared Variable	Use of a variable not present in declarations
Uninitialized Variable	Reading a variable before assignment
Type Mismatch	Assignment or operation incompatible with the expected type
Unknown Function Call	Invocation of a function that is undefined or misspelled
Incorrect Number of Arguments	Function call with too many or too few parameters
Out-of-Bounds Array Access	Attempt to access an index that does not exist
Unsupported Type	Use of a non-existent or malformed type
Invalid Expression	Incorrect syntax in an assignment or a conditional test
User Input Error	Input of a value that does not conform to the variable's type
Invalid Instruction	Unrecognized keyword or structure

Examples of Errors and Corresponding Messages

Undeclared Variable

```
Algo
Begin
    x <-- 5  # x has not been declared
End
```

```
RunTime error: Variable 'x' is not declared

    x <-- 5  # x has not been declared
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Type Mismatch

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

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

    x <-- "text"
          ^^^^^^^
```

Unknown Function Call

```
Algo
    result : int
Begin
    result <-- unknown(3)
End
```

```
RunTime error: 'unknown' is not defined

    result <-- unknown(3)
                 ^^^^^^^^^^
```

Incorrect Number of Arguments

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

Algo
Begin
    print f(3)  # one argument is missing
End
```

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

    print f(3)  # one argument is missing
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Out-of-Bounds Array Access

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

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

    print tab[5]  # out of bounds
          ^^^^^^^^^^^^^^^^^^^^^
```

Error Triggering

Errors are triggered either:

- **During static analysis** (declarations, types)
- **At runtime** (memory access, dynamic calls, user input)

The interpreter halts execution as soon as an error is detected, indicating precisely **the line concerned**, **the type of error**, and **the variable or function involved**.

Best Practices to Avoid Errors

- Always declare variables with their type before ~Begin~.
- Respect types in assignments and function calls.
- Verify array dimensions before any index access.
- Read error messages carefully; they are designed to be explicit.

Chapter 5 : Examples

Examples DAUPS

Example 1

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

Example 2

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

Example 3

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

Example 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
```

Example 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
```

Example 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 <=> 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)!=0)
        x <-- ((b*f)-(e*c))/((d*b)-(e*a))
        if b == 0 then
            # e!=0
            y <-- (f/e)-(d/e)*x
        else
            y <-- (c/b)-(a/b)*x
        print "x=", x
        print "y=", y
End
```

Example 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
```

Example 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 etre 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
```

Example 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
```

Example 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
```

Example 11

```
Algo
# Tous les nombres parfaits ≤ 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
```

Example 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
```

Example 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
```

Example 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
```

Example 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 im
End
```

Example 16

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

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

Example 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
```

Example 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

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
```

Example 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
```

Example 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
```

Example 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
```

Example 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-lig
    tab <-- tab2

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

Example 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
```

Example 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
```

Example 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
```

Example 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
            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
```

Chapter 6 : Execution

Execution Flow in the DAUPS Language

The DAUPS interpreter executes a program following a rigid, statically-typed structure in which blocks, variables, functions, and instructions are validated both statically and dynamically.

This chapter describes the complete execution pipeline, from loading a `.daups` file to producing the results.

Main Execution Steps

1. Loading the source file
2. Lexical Analysis / Tokenization
3. Syntax Analysis
4. Semantic Analysis (types, functions, scopes…)
5. Symbol Table Construction
6. Line-by-Line Execution
7. Error Handling and Result Display

1. Loading the Source File

DAUPS source files are typically `.daups` or `.txt` files containing a structure such as:

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

The file is read in its entirety, and then each line is processed sequentially.

2. Syntax Analysis and Block Construction

The interpreter identifies the main blocks (`Algo`, `Begin`, `End`) and builds a logical representation of the program, including:

- Simple instructions (`get`, `print`, `x <-- 3`)
- Control blocks (`if`, `while`, `for`)
- Function definitions
- Variable declarations

3. Symbol Table

Declared variables and defined functions are recorded in a symbol table:

- Globally for the entire program
- Locally for each function's scope

Each symbol is associated with:

- a name
- a type (`int`, `float`, etc.)
- a value (initial or determined at runtime)
- a context (local or global)

4. Instruction Execution

The main block is executed in order of appearance. Each line may correspond to one of the following:

- an assignment
- a user input instruction
- a control structure
- a function call (built-in or user-defined)

Expressions are evaluated dynamically with type checking.

5. Function Calls

When a function is called:

- Arguments are evaluated
- A local context is created
- Local variables are isolated from the global environment
- A value is returned if the function specifies one

Example:

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

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

6. Error Handling

Any error detected during execution (undeclared variable, type mismatch, index out of bounds, etc.) immediately halts the program with an explicit error message.

7. End of Execution

When all instructions in the main block have been executed:

- Results are displayed in the console (or captured if redirected)
- The program terminates normally if no error was raised

Example of a Complete Executed Program

```
Algo
    a, b : int
    result : int
Begin
    get a
    get b
    result <-- a + b
    print "Result :", result
End
```

Summary

Execution in DAUPS follows a strict but predictable structure:

- No compilation: everything is interpreted dynamically
- Types and scopes are strictly enforced
- Error checking is systematic
- Functions and arrays are managed dynamically

Made by PerseusShade