Țibuleac Gabriela
Grupa E3

PyScal Interpreter

Language Specification & Design Report

Cuprins

---

1. Introducere

1.1. Prezentare Generală

Acest proiect prezintă designul și implementarea unui interpretor pentru un limbaj de programare numit Pyscal. Limbajul a fost conceput în scop educațional, pentru a ilustra conceptele fundamentale din limbajul Python.

Obiectivul principal a fost crearea unui limbaj care îmbină claritatea sintaxei moderne, inspirată din Python, cu rigoarea structurilor de control clasice, blocuri delimitate explicit.

1.2. Obiectivele Proiectului

Implementarea acestui interpretor a urmărit atingerea următoarelor obiective tehnice:

- Analiză Lexicală:

  Transformarea codului sursă într-un flux de tokeni, gestionând identificatori, literali și cuvinte cheie.

- Parsing Recursiv:

  Implementarea unui recursive descent parser pentru a valida sintaxa și a construi arborele sintactic abstract (AST).

- Execuție Interpretată:
  Parcurgerea arborelui sintactic abstract pentru a executa instrucțiunile.

- Extensibilitate:

  Design modular care permite adăugarea ușoară de noi funcționalități.

1.3. Caracteristici Principale

Limbajul propus se distinge prin următoarele funcționalități cheie:

- Sistemul de Tipuri Dinamic:

  Variabilele nu necesită declararea tipului, oferind flexibilitate programului, suportă numere, string-uri, liste.

- Scoping Lexical:

  Implementarea unui system robust de medii de execuție, Environments, care permite vizibilitatea corectă a variabilelor.

- Funcții First-Class:
  Funcțiile sunt tratate ca valori, permițând concepte precum Closures, funcții care capturează mediul, și Higher-Order Functions.

- Structuri de control:

  Suport complet pentru decizii, if/else, și iterații, while, for, folosind blocuri explicite, BEGIN ... END, pentru a evita ambiguitățile sintactice.

2. Arhitectura Sistemului

Arhitectura interpretorului urmează un model modular clasic, divizat în trei etape principale de procesare. Această structură asigură o separare a responsabilităților și facilitează depanarea și extinderea limbajului.

Fluxul de execuție este secvențial:

Source Code → Lexer → Parser → AST → Interpreter → Output

2.1. Analiza Lexicală

- Fișier: lexer.py

Lexerul transformă codul sursă într-o listă de tokeni. În implementarea clasei Lexer, metoda principală tokenezi() parcurge textul și delegă procesarea către metode specializare în funcție de caracterul întâlnit:

- read_number():

  Detectează cifre și construiește numere întregi sau float, prin verificarea punctului zecimal.

- read_string():
  Extrage textul dintre ghilimele “ sau apostrof ‘.

- read_identifier():

Identifică variabile sau cuvinte cheie. Dacă textul găsit există în dicționarul KEYWORDS, este marcat ca atare, astfel, devine un IDENTIFIER.

2.2. Analiza Sintactică

- Fișier: parser.py

Parserul primește lista de tokeni și verifică dacă aceștia respectă gramatica limbajului. Am utilizat recursive descent parsing, unde fiecare regulă gramatică are o metodă corespondentă în clasa Parse.

Pentru a respecta prioritatea operatorilor matematici, metodele sunt stratificate:

- parse_expression():
  Apelează parse_logic().

- parse_term():
  Gestionează + și -.

- parse_factor():
  Gestionează *, / și %.

Această ierarhie asigură că o expresie de tipul 2+3*4 este parsat corect ca 2+(3*4).

2.3. Reprezentarea Intermediară. AST Nodes

- Fișier: nodes.py

Nodurile AST sunt clase simple care se moștenesc din clasa de bază ASTnode. Acestea conțin doar date, precum:

- BinOp:
  Reține left, op, right.

- If:
  Reține condition, then_branch, lista de instrucțiuni și else_branch.

- FunctionDef:
  Stochează numele funției, parametri și corpul.

2.4. Interpretorul

Interpretorul parcurge arborele AST folosind pattern-ul visitor. Am separat execuția în două metode principale:

- evaluate(expr):
  Pentru expresii care returnează o valoarea, BinOp, Number, Call.

- execute(stmt):
  Pentru instrucțiuni care modifică starea programului, Assign, If, While.

Pentru structurile repetitive, interpretorul foloseste bucle native din Python. De exemplu, în metoda visit_While, interpretorul verifcă condiția și execute corpul buclei atâta timp cât evaluate(node.condition) este adevărată.

2.5. Gestionarea Mediului. Environment

- Fișier: envi.py

Clasa Environment este esențială pentru Lexical Scoping. Fiecare mediu are un atribut vars, dicționar pentru variabile, și un atribut parent, referință către mediul părinte.

Metoda get(name) implementează căutarea recursivă:

- Caută variabila în mediul current self.vars.

- Dacă nu găsește și self.parent există, apelează parent.get(name).

- Dacă nu găsește nicăieri, ridică NameError.

3. Specificația Limbajului. Sintaxă. Semantică.

3.1. Sistemul de Tipuri

Limbajul utilizează Tipizare Dinamică. Variabilele nu au un tip declarant static. Tipul este asociat valorii, nu numelui variabilei. Verificarea tipurilor se realizează la runtime, apelam RuntimeError în caz de incompatibilitate.

Tipurile de date fundamentale suportate, implementarea în nodes.py și gestionează în interpreter.py:

TIP DE DATE	DESCRIERE	EXEMPLU COD	REPREZENTARE INTERNĂ
Number	Numere întregi și reale	x=10, y=3.12	int, float
String	Șiruri de caractere imutabile	s=”Bună”	str
List	Șiruri ordonate, mutabile	v=[1,2,” a”]	list
Boolean	Valori de adevăr, rezultat logic	1, 0	int/bool
Function	Funcții ca obiecte	def f(x): ..	Instanță de clasa Function

3.2. Gramatică Formală

Sintaxa limbajului este procesată de un parser recursiv, care impune reguli stricte de organizare a codului. În loc să folosim notații matematice complexe (BNF), vom descrie structura gramaticală prin regulile logice aplicate.

3.2.1. Structura Programului și Instrucțiuni

Un program este o secvență liniară de instrucțiuni, executate de sus în jos. Instrucțiunile pot fi simple (o singură linie) sau compuse (blocuri de cod).

- Atribuirea:

Se realizează folosind semnul egal (=). Identificatorul din stânga primește valoarea expresiei din dreapta (ex: scor = 100).

- Afișarea:

Se utilizează instrucțiunea PRINT urmată de o expresie între paranteze.

- Returnarea valorilor:

Cuvântul cheie RETURN oprește execuția unei funcții și trimite o valoare înapoi apelantului.

3.2.2. Blocuri de Cod

Un element distinctiv al limbajului, inspirat din Pascal, este delimitarea explicită a blocurilor de cod.

- Pentru a grupa mai multe instrucțiuni (de exemplu, în corpul unei bucle while sau al unei funcții), se utilizează cuvintele cheie BEGIN și END.

- Această abordare elimină ambiguitățile cauzate de indentare și simplifică citirea codului, oferind limite vizuale clare pentru fiecare scope.

3.2.3. Structuri de Control

Parserul recunoaște trei structuri fundamentale pentru controlul fluxului:

1. Instrucțiunea IF:

Evaluează o condiție logică. Dacă este adevărată, execută ramura principală; opțional, poate avea o ramură ELSE pentru cazul contrar.

2. Bucla WHILE:

Execută repetitiv un bloc de cod atâta timp cât condiția specificată este adevărată. Este ideală pentru algoritmi unde numărul de pași nu este cunoscut dinainte.

3. Bucla FOR**:**

O structură iterativă clasică, definită prin: variabila contor, valoarea de start și valoarea de final (separate prin cuvântul cheie TO).

3.2.4. Definirea Funcțiilor

Funcțiile sunt definite folosind cuvântul cheie def, urmat de numele funcției și lista de parametri între paranteze. Corpul funcției este separat de antet prin caracterul două puncte (:). Datorită suportului pentru First-Class Functions, funcțiile pot fi imbricate (funcții definite în interiorul altor funcții).

3.2.5. Expresii și Prioritatea Operatorilor

Expresiile matematice și logice sunt construite respectând ordinea naturală a operațiilor. Parserul este stratificat pentru a asigura corectitudinea matematică:

1. Prioritate Maximă**:**

Parantezele (), apelurile de funcții și accesarea elementelor din liste [ ].

2. Factori**:**

Înmulțirea (*), împărțirea (/) și restul împărțirii (%).

3. Termeni**:**

Adunarea (+) și scăderea (-).

4. Comparații**:**

Operatorii relaționali (==, <, >, etc.).

5. Logic**:**

Operatorii logici (AND, OR), care au cea mai mică prioritate.

3.3. Semantică. Reguli de Execuție

3.3.1. Scoping

Limbajul implementează Lexical Scoping.Rezolvarea variabilelor se face cautând numele în mediul current Environment.vars.Dacă nu este găsit, căutarea continuă recursive în mediul părinte Environment.parent.Dacă se ajunge la mediul global și variabila nu există, se va afișa eroarea NameError.

3.3.2. Evaluarea Expresiilor

Operatorii aritmetici respectă ordinea matematică standard. Listele sunt indexate de la 0. Accesarea unui index invalid generează RuntimeError. Argumentele funcțiilor sunt evaluate prin valoare înainte de executarea corpului funcției.

3.4. Funcții Built-in

Interpretorul oferă un set de funcții predefinite, implementate direct în Python pentru performanță:

- PRINT(expr): Evaluează expresia și afișează rezultatul la console.
- INPUT(): Citește o linie text de la tastatură.
- LEN(collection): Returnează lungimea unui string sau a unei liste.
- TO_INT(expr): Convertește un string numeric în integer.
- TO_STR(expr): Convertește orice valoare în string.

4. Caracteristici Avansate și Algoritmi

4.1. Funcții First-Class

În limbaj, funcțiile sunt tratate ca first-class. Aceasta înseamnă că ele nu sunt doar structuri statice de cod, ci valori propriu-zise, obiecte care pot fi manipulate la fel ca numerele sau șirurile de caractere. Sistemul permite:

- Atribuirea funcțiilor la variabile:

O funcție poate fi redenumită sau stocată într-o variabilă.

- Pasarea ca argumente:

O funcție poate fi trimisă ca parametru unei alte funcții.

- Returnarea ca rezultat:

O funcție poate crea și returna o altă funcție.

Această funcționalitate permite scrierea de funcții de ordin superior, oferind programului o putere de abstratizare simalară cu cea a limbajului Python.

4.2. Closures

Closure apare atunci când o funcție interioară capturează și reține accesul la variabilele din mediul funcției exterioare, chiar și după ce funcția exterioară și-a încheiat execuția.

Implementare :

Mecanismul se bazează pe modul în care clasa Environment gestionează ierarhia mediilor în fișierul envi.py.

- Atunci când o funcție este definită (la execuția nodului FunctionDef), interpretorul salvează în obiectul funcției o referință către mediul curent (self.env). Aceasta reprezintă "memoria" funcției la momentul nașterii ei.

- Când funcția este apelată ulterior, corpul ei este executat într-un mediu nou, dar interpretorul setează ca părinte al acestui nou mediu chiar mediul salvat anterior (closure-ul), și nu mediul din care se face apelul.

- Astfel, funcția are acces garantat la variabilele care existau în jurul ei la definire, asigurând persistența datelor și izolarea corectă a variabilelor.

4.3. Recursivitate

Limbajul suportă complet recursivitatea, permițând definirea de algoritmi care se auto-apelează pentru a rezolva sub-probleme (exemplu clasic: calculul factorialului sau generarea șirului lui Fibonacci).

Mecanismul de funcționare: Recursivitatea funcționează natural datorită mecanismului de Lexical Scoping. Când o funcție se auto-apelează în interiorul propriului corp, interpretorul trebuie să rezolve numele funcției. Deoarece funcția a fost deja definită în mediul exterior (înainte de execuția corpului), ea este vizibilă pentru ea însăși. Fiecare apel recursiv generează o nouă instanță a clasei Environment. Aceste medii se stivuiesc, fiecare având propriile variabile locale, ceea ce asigură că starea fiecărui pas recursiv (valoarea parametrilor la pasul n) este izolată și nu interferează cu pasul n-1.

5. Raport Decizii de Design

5.1. Alegerea Limbajului de Implementare

Am ales ca limbaj de bază Python din următoarele motive:

- Productivitate: Python permite prototiperea rapidă a structurilor de date complexe, precurm arbori AST sau tabele de simboluri.
- Lizibilitate: Codul sursă este ușor de urmărit, pemițând concentrarea pe logica algoritmilor și nu pe detaliile de implementare.
- Educativ: Am aprofundat partea de gestionarea erorilor, ordonarea codului cât să fie ușor de înțeles.

5.2. Arhitectura

Pentru execuția codului, am ales varianta de interpretor Tree-Walk ,care parcurge direct arborele sintactic, în loc să generez. Este destul de intuitiv. Când ceva nu mergea, puteam să văd exact în ce nod al arborelui m-am blocat. Am vrut să văd legătura directă dintre codul scris (if x > 5) și execuția lui în Python, iar modelul Tree-Walk face exact asta: vizitează fiecare nod și îl execută pe loc.

5.3. Parserul

Am decis să nu folosesc indentarea pentru blocuri ca în Python, ci să folosesc cuvintele cheie BEGIN ... END (stil Pascal). Implementarea indentării în Lexer este dificilă și predispusă la erori. Folosind delimitatori expliciți, codul este mult mai clar și parserul știe exact când se termină un bloc while sau o funcție. Asta a simplificat mult logica de citire a codului.

5.4. Scoping

Am ales varianta Environment Chaining, pentru că voiam neapărat să suport Recursivitate și Closures. A trebuit să fac funcțiile să țină minte mediul în care au fost create. Variabilele sunt rezolvate pe baza structurii codului, nu a stivei de apeluri runtime.

5.5. Bucla For

Am stabilit utilizarea sintaxei în stil Pascal (variabila = start to final) pentru bucla For, spre deosebire de varianta Python cu range(), datorită clarității explicite a limitelor, oferind o structură intuitivă și simplificată pentru începători.

Pycal User Documentation

Cuprins

---

6. Introducere

6.1. Prezentare Generală

Acest proiect prezintă designul și implementarea unui interpretor pentru un limbaj de programare numit Pyscal. Limbajul a fost conceput în scop educațional, pentru a ilustra conceptele fundamentale din limbajul Python.

6.2. Ce vei învăța?

La finalul acestui ghid vei putea:

- Scrie și rula programe simple

- Lucra cu variabile și tipuri de date
- Crea funcții și algoritmi
- Rezolva probleme practice de programare

7. Instalare și Configurare

7.1. Cerințe de Sistem

- Ai nevoie să ai instalat Python cu o versiune egală sau mai mare de 3.0.
- Editor de text (VS Code, Notepad++, sau orice alt editor)
- Terminal/Command Prompt

7.2. Structura Fișierelor

Proiectul conține următoarele fișiere:

proiect	Numele proiectului
lexer.py	Analiza lexicală
parser.py	Analiza sintactică
interpreter.py	Executorul de cod
nodes.py	Structuri AST
envi.py	Gestionarea variabilelor
errors.py	Gestionarea erorilor
test.py	Exemple de cod

7.3. Rulare

Pentru a executa interpretorul și a rula suita de teste integrată, deschideți terminalul în directorul proiectului și rulați comanda:

python test.py

8. Feedback Vizual

Utilizez un sistem de marcare cromatică a ieșirii în consolă. Acest capitol explică semnificația fiecărei culori utilizate în terminal, ajutând utilizatorul să identifice rapid starea execuției.

8.1. Tabel de Semnificație a Culorilor

Sistemul folosește coduri de evadare ANSI definite în clasa Colors pentru a structura informația.

Culoare	Semnificație	Context de Utilizare
CYAN	Informare / Header	Folosit pentru titlurile testelor ([TEST] Descriere) și secțiunile majore, pentru a le distinge de log-ul efectiv.

GALBEN	Categorii / Atenționări	Evidențiază categoriile mari de teste (ex: >>> SINTAXA <<<) și instrucțiunile interactive (ex: >>> ATENTIE: Introdu valori...).
GRI	Cod Sursă	Afișează codul Pyscal care este executat în momentul respectiv. Culoarea ștearsă ajută utilizatorul să se concentreze pe rezultat, nu pe codul sursă.
VERDE	Succes	Indică faptul că un test a fost trecut cu brio sau că o operație s-a încheiat conform așteptărilor (ex: SUCCESS).
ROȘU	Eroare/ Eșec	Semnalează o problemă critică. Este folosit pentru mesajele de eroare (RuntimeError, SyntaxError) și pentru testele care au eșuat (FAILED).

8.2. Interpretarea Output-ului Colorat

8.2.1. Execuție cu Succes

Când rulați suita de teste, un rezultat pozitiv va arăta astfel:

- Titlul testului apare în CYAN.
- Codul rulat apare în GRI.
- Mesajul final SUCCESS apare în VERDE.

8.2.2. Raportarea Erorilor

Sistemul de erori (errors.py) este integrat cu acest modul de culori. Orice excepție ridicată de interpretor (fie că este de sintaxă sau de execuție) va fi afișată automat în ROȘU pentru a atrage imediat atenția utilizatorului asupra liniei problematice.

Exemplu vizual de eroare:
Eroare Runtime [Linia 3]: Impartire la 0

9. First Try

9.1. Hello, World!

Primul tău program:
Input:
PRINT("Hello, World!")
Output:
Hello, World!

9.2. Comentarii

Comentariile încep cu # și sunt ignorate de interpretor:
Input:

Acesta este un comentariu
PRINT("Salut!") # Comentariu la sfârșitul liniei
Output:
Salut!

9.3. Afișarea Mesajelor
Folosește PRINT() pentru a afișa text:
Input:
PRINT("Bine ai venit!")
Output:
Bine ai venit!

10. Tipuri de Date
Limbajul este dinamic. Nu este necesară declararea tipurilor.
- Tipuri suportate: Integer, Float, String, List, Function.
- Atribuire: x = 10 sau nume = "Text".

10.1. Numere Întregi
Input:
varsta = 25
PRINT(varsta)
Output:
25

10.2. Numere Zecimale (Float)
temperatura = 36.6
PRINT(temperatura)
Output:
36.6

10.3. Șiruri de Caractere (String)
Text între ghilimele. Poți folosi ghilimele duble " sau simple '.
Input:
nume = "Ana"
PRINT(nume)
Output:
Ana

10.4. Liste
Input:
numere = [1, 2, 3, 4, 5]
PRINT(numere)
Output:
[1, 2, 3, 4, 5]

10.5. Valori Boolean

Input:

este_student = true

PRINT(este_student)

Output:

True

11. Variabile și Operații

11.1. Declararea Variabilelor

Limbajul este dinamic. Nu este necesară declararea tipurilor. Variabilele se creează prin atribuire.

Input:

```
# Numere
x = 10
y = 3.14

# Text
nume = "Maria"
prenume = "Popescu"

# Liste
note = [8, 9, 10, 7]
```

11.2. Operații Aritmetice

Input:

```
a = 10
b = 3

# Adunare
suma = a + b
PRINT(suma)
```

Output:

13

Input:

```
# Scădere
diferenta = a - b
PRINT(diferenta)
```

Output: 7

Input:

# Înmulțire
produs = a * b
PRINT(produs)
Output: 30

Input:
# Împărțire
cat = a / b
PRINT(cat)
Output: 3.333...

Input:
# Modulo (rest)
rest = a % b
PRINT(rest)
Output: 1

11.3. Concatenare de String-uri
Operatorul + unește string-uri:
Input:
prenume = "Ion"
nume = "Popescu"

# Concatenare
nume_complet = prenume + " " + nume
PRINT(nume_complet)
Output:
Ion Popescu

Input:
# Concatenare cu numere (conversie automată)
varsta = 25
mesaj = "Am " + varsta + " ani"
PRINT(mesaj)
Output:
Am 25 ani

11.4. Operatori de Comparație
x = 10
y = 20

Mai mic

```
PRINT(x < y)   # Output: True

# Mai mare
PRINT(x > y)   # Output: False

# Egal
PRINT(x == y)  # Output: False

# Diferit
PRINT(x != y)  # Output: True

# Mai mic sau egal
PRINT(x <= 10) # Output: True

# Mai mare sau egal
PRINT(y >= 20) # Output: True
```

11.5. Operatori Logici

```
# AND - adevărat doar dacă ambele sunt adevărate
PRINT(true AND true)   # Output: True
PRINT(true AND false)  # Output: False

# OR - adevărat dacă cel puțin una este adevărată
PRINT(true OR false)   # Output: True
PRINT(false OR false)  # Output: False

# Combinații
x = 15
PRINT(x > 10 AND x < 20)  # Output: True
PRINT(x < 5 OR x > 100)   # Output: False
```

12. Structuri de Control

12.1. Instrucțiunea IF

12.1.1. IF simplu

Input:

```
varsta = 18
if varsta >= 18:
  PRINT("Ești major")
```

Ouput:

Ești major

12.1.2. IF-ELSE

Input:

```
temperatura = 15
if temperatura > 20:
   PRINT("Este cald afară")
else:
   PRINT("Este răcoare afară")
```

Output:

Este răcoare afară

12.1.3. IF cu Blocuri BEGIN-END

Input:

```
nota = 9
if nota >= 5: BEGIN
   PRINT("Ai promovat!")
   PRINT("Felicitări!")
   media = nota
END
else: BEGIN
   PRINT("Nu ai promovat")
   PRINT("Mai încearcă")
END
```

12.2. Bucla While

12.2.1. While Simplu

Input:

```
nota = 9
if nota >= 5: BEGIN
   PRINT("Ai promovat!")
   PRINT("Felicitări!")
   media = nota
END
else: BEGIN
   PRINT("Nu ai promovat")
   PRINT("Mai încearcă")
END
```

Output:

1
2

```
3
4
5
```

12.2.2. WHILE cu Bloc

```
Input:
contor = 0
suma = 0
while contor < 5: BEGIN
   suma = suma + contor
   PRINT("Contor: " + TO_STR(contor) + ", Suma: " + TO_STR(suma))
   contor = contor + 1
END
Output:
Contor: 0, Suma: 0
Contor: 1, Suma: 1
Contor: 2, Suma: 3
Contor: 3, Suma: 6
Contor: 4, Suma: 10
```

12.3. Bucla FOR

```
Input:
for numar = 1 to 10: BEGIN
   patrat = numar * numar
   PRINT(TO_STR(numar) + " la patrat = " + TO_STR(patrat))
END
Output:
1 la patrat = 1
2 la patrat = 4
3 la patrat = 9
4 la patrat = 16
5 la patrat = 25
6 la patrat = 36
7 la patrat = 49
8 la patrat = 64
9 la patrat = 81
10 la patrat = 100
```

13. Funcții

13.1. Funcție Simplă

Input:

```
def saluta():
    PRINT("Salut!")
# Apelare
saluta()
```

Ouput:

```
Salut!
```

13.2. Funcție cu Parametri

Input:

```
def saluta_persoana(nume):
    PRINT("Salut, " + nume + "!")
saluta_persoana("Ana")
saluta_persoana("Ion")
```

Output:

```
Salut, Ana!
Salut, Ion!
```

13.3. Recursivitate

Input:

```
def factorial(n): BEGIN
    if n <= 1:
        return 1
    else:
        return n * factorial(n - 1)
END

PRINT(factorial(5))  # Output: 120
PRINT(factorial(6))  # Output: 720
```

13.4. Closures

```
def creeaza_inmultitor(factor): BEGIN
    def inmulteste(x):
        return x * factor
    return inmulteste
END

de_2_ori = creeaza_inmultitor(2)
de_5_ori = creeaza_inmultitor(5)

PRINT(de_2_ori(10))  # Output: 20
PRINT(de_5_ori(10))  # Output: 50
```

13.5. Higher-Order Functions

Input:

```
def aplica_operatie(functie, valoare): BEGIN
   rezultat = functie(valoare)
   return rezultat
END

def dublu(x):
   return x * 2

def patrat(x):
   return x * x

PRINT(aplica_operatie(dublu, 5))   # Output: 10
PRINT(aplica_operatie(patrat, 5))  # Output: 25
```

14. Liste

14.1. Creare Listelor

```
# Liste goale
lista_goala = []

# Liste cu elemente
numere = [1, 2, 3, 4, 5]
fructe = ["mar", "banana", "portocala"]
mixt = [10, "text", 3.14]
```

14.2. Accesarea Elementelor

Indexarea începe de la 0:

```
fructe = ["mar", "banana", "portocala", "struguri"]
primul = fructe[0]
al_doilea = fructe[1]
ultimul = fructe[3]
PRINT(primul)     # Output: mar
PRINT(al_doilea)  # Output: banana
PRINT(ultimul)    # Output: struguri
```

15. Funcții Built-in

15.1. Print()

Afișează valori la consolă:

```
PRINT("Text")
PRINT(42)
PRINT(3.14)
PRINT([1, 2, 3])
```

15.2. INPUT()

Citește input de la utilizator:

```
PRINT("Cum te cheamă?")
nume = INPUT()
PRINT("Salut, " + nume + "!")
```

Output:

Cum te cheamă?

> Ana (scriem în terminal)

Salut, Ana!

15.3. LEN()

Returnează lungimea unui string sau a unei liste:

```
# Pentru string-uri
text = "Salut"
lungime_text = LEN(text)
PRINT(lungime_text)  # Output: 5
# Pentru liste
lista = [1, 2, 3, 4]
lungime_lista = LEN(lista)
PRINT(lungime_lista)  # Output: 4
```

15.4. TO_INT()

Convertește la număr întreg:

```
text_numar = "42"
numar = TO_INT(text_numar)
rezultat = numar + 8
PRINT(rezultat)  # Output: 50

# Funcționează și cu float
zecimal = 3.14
intreg = TO_INT(zecimal)
PRINT(intreg)  # Output: 3
```

15.5. TO_STR()

Convertește orice valoare la string:

```
numar = 100
text = TO_STR(numar)
mesaj = "Numărul este " + text
PRINT(mesaj)  # Output: Numărul este 100
```

```
# Pentru liste
lista = [1, 2, 3]
lista_text = TO_STR(lista)
PRINT(lista_text)  # Output: [1, 2, 3]
```

16. Depanare și Erori Comune
    16.1. Tipuri de Erori
        16.1.1. Erori Lexicale
            - Caracter Ilegal: x = 5 @ 3

            Eroare**:** Caracter Ilegal: Caracterul '@' nu este permis

            - String Neterminat:
              mesaj = "Salut
              PRINT(mesaj)

            Eroare: String neinchis (lipseste ghilimea de inchidere)

        16.1.2. Erori de Sintaxă
        PRINT("Salut"
        Eroare: Eroare de sintaxa: Ma asteptam la RPAREN

        16.1.3. Erori Runtime
            - Împărțire la Zero: x = 10 / 0

            Eroare**:** Eroare Runtime: Impartire la 0

            - Variabilă Nedefinită: PRINT(variabila_inexistenta)

            Eroare: Eroare de Nume: Variabila 'variabila_inexistenta' nu este definite

            - Index în Afara Limitelor:
              lista = [1, 2, 3]
              PRINT(lista[10])
              Eroare Runtime: Index in afara limitelor: 10 (lungime lista: 3)

            - Eroare Nedefinită
              rezultat = functie_inexistenta(5)
              Eroare: Eroare Runtime: Functie nedefinita: 'functie_inexistenta'1

    16.2. Sfaturi pentru Depanare

Mesajele includ numărul de linie și coloană:
Eroare Runtime [Linia 5, Coloana 10]: Variabila 'x' nu este definita