Universitatea *Transilvania* din Braşov
Facultatea de Matematică şi Informatică
Departamentul de Matematică şi Informatică

Construcția unui Interpretor

Autor: Vaida Raluca-Maria
Informatică Aplicată, Anul II
Grupa 10LF333

2025

Interpretor

Cuprins

1 Generalități

În cadrul acestei lucrări va fi prezentat design-ul și implementarea unui mini-interpretor pentru un subset de funcționalități de tip C/C++. Această lucrare are ca scop sumarizarea procesului de dezvoltare a unui astfel de interpreter. De asemenea, se va focusa și pe relevanța existenței interpretoarelor.

Scopul unui interpretor este de a traduce în cod mașină un program sursă scris într-un limbaj de nivel înalt. Limbajul folosit pentru a concepe un interpretor se numește limbaj de implementare.

Diferența esențială între un interpretor și un compilator este faptul că în timp ce un compilator generează cod obiect scris în cod mașină, un interpretor executa instrucțiuni.

1.1 Procesul de interpretare a uni cod sursă

Procesul de interpretare a unui cod sursă este împărțit în 6 incremente funcționale. Acestea sunt:

- framework-ul
- scanner-ul
- tabela de simboluri
- parsarea și interpretarea expresiilor și a instrucțiunilor de atribuire
- parsarea și interpretarea instrucțiunilor de control
- arsarea și interpretarea declarărilor

În Figura 2 este ilustrat faptuul că front end-ul este folosit atât de compilator cât și de interpretor. Acesta se ocupă cu citirea codului sursă și faza inițială de traducere. Componentele principale ale acestuia sunt: parser-ul, scanner-ul, token-ul și sursa.

Parser-ul este cel care este în continuă așteptare de noi token, pe care îi primește de la scanner. Acesta asociază token-ul unui element de limbaj înalt a cărui funcționalitate o mimează. Totodată parser-ul e cel care verifică dacă codul este corect din punct de vedere sintactic. Scanner-ul citește caracterele din codul sursă și îl împarte în tokeni.

Un exemplu pentru a înțelege scanarea:

TYPE	TEXT STRING
Word (reserved word)	int
Word (identifier)	a
Special symbol	=
Number (integer)	3
Special symbol	;

Figura 1: Procesul de scanare

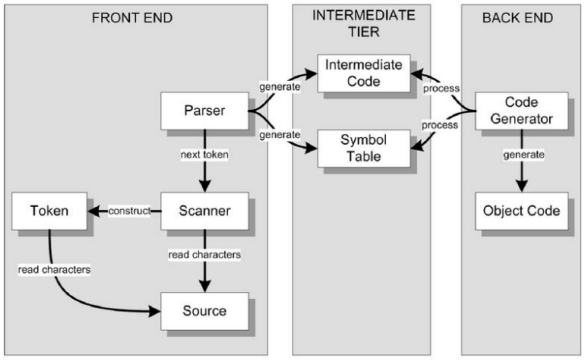

Figura 2: Design-ul conceptual al unui interpretor

Sintaxa unui limbaj de programare este determinată de un set fix de reguli gramaticale care verifică dacă o expresie corespunde limbajului curent.

1.2 Abstract Syntax Tree

Un AST (Abstract Syntax Tree) este o reprezentare structurală a codului sursă, organizată sub forma unui arbore, unde fiecare nod simbolizează o entitate sintactică (cum ar fi o expresie, o instrucțiune sau un bloc de cod). În contextul dezvoltării unui interpretor, AST joacă un rol esențial, fiind interfața dintre analizarea codului și execuția sa.

Procesul începe cu parser-ul, care construiește AST-ul pe baza regulilor definite în gramatică. AST-ul simplifică structura codului, eliminând elementele redundante (cum ar fi punctuația) și păstrând doar informațiile esențiale pentru procesare.

Interpretorul traversează AST-ul pentru a executa codul. În această etapă, fiecare nod al arborelui este vizitat și procesat conform logicii implementate. De exemplu, un nod care reprezintă o instrucțiune de atribuire actualizează o variabilă în scope-ul curent, iar un nod pentru o expresie aritmetică evaluează operandul stâng și drept, aplicând operatorul.

AST-ul facilitează suportul pentru funcționalități avansate, precum funcțiile, structurile de control (if, for) și recursivitatea, deoarece permite un model ierarhic care reflectă relațiile logice din cod. În plus, separă clar analiza sintactică de execuție, ceea ce face interpretorul mai modular și ușor de extins. De asemenea, AST este crucial pentru validarea regulilor de sintaxă și verificarea semantică înainte de execuție, oferind o bază solidă pentru dezvoltarea interpretatorului.

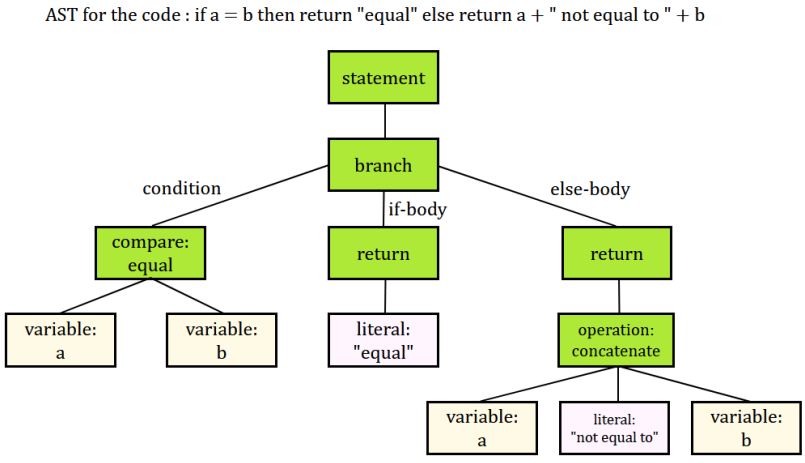

Figura 3: Exemplu conceptual de AST

2 Dezvoltarea unui interpretor

Pentru dezvoltarea acestui interpretor a fost ales generatorul de parsare ANTLR, bazat pe algoritmi LL și limbajul de programare C#.

2.1 Construirea gramaticii

Primul pas este construirea gramaticii cu ajutorul IDE-ului Visual Studio Code într-un fișier cu extensia .g4. O gramatică cuprinde reguli de lexer și reguli de parser.

Regulile unui lexer reprezintă setul de cuvinte-cheie și expresii regulate pe care limbajul le recunoaște. Aceste reguli permit identificarea elementelor esențiale din cod, cum ar fi cuvintele rezervate (din limbaje de nivel înalt) sau tipurile de date și valorile suportate de acestea, folosind expresii regulate pentru o recunoaștere flexibilă și adaptată nevoilor limbajului conceput.

```
INTEGER_TYPE: 'int';
FLOAT_TYPE: 'float';
STRING_TYPE: 'string';
DOUBLE_TYPE: 'double';
VOID_TYPE: 'void';
IF_KEYWORD: 'if';
ELSE_KEYWORD: 'else';
WHILE_KEYWORD: 'while';
FOR_KEYWORD: 'for';
RETURN_KEYWORD: 'return';
PRINT_KEYWORD: 'Print';
```

Figura 4: Reguli de lexer

```
INTEGER_VALUE: ('0' | [1-9][0-9]*);
FLOAT_VALUE: ('0' | [1-9][0-9]*) '.' [0-9]+;
STRING_VALUE: '"' (~["\\] | '\\' .)* '"';
DOUBLE_VALUE: FLOAT_VALUE [eE] [+-]? [0-9]+;
```

Figura 5: Reguli de lexer

Regulile unui parser se scriu de regulă cu litere mici și definesc modul în care structura sintactică a unui program este analizată și validată conform gramaticii limbajului. Ele determină relațiile dintre elementele de bază (token-uri) identificate de lexer, pentru a forma expresii, instrucțiuni și blocuri logice valide.

```
functionDefinition:
    type VARIABLE_NAME LPAREN parameterList? RPAREN LBRACE statement* RBRACE;

parameterList: parameter (COMMA parameter)*;
parameter: type VARIABLE_NAME;

functionCall: VARIABLE_NAME LPAREN argumentList? RPAREN;
argumentList: expr (COMMA expr)*;
```

Figura 6: Reguli de parser

```
statement:
    declaration
    | printStmt
    | assignmentStmt
    | exprStmt
    | blockStmt
    | returnStmt;

blockStmt: ifStmt | whileStmt | forStmt;
```

Figura 7: Reguli de parser

Utilizând ANTLR, se generează automat codul sursă pentru lexer și parser în limbajul C#. Lexer-ul identifică token-urile (unitățile lexicale) conform regulilor definite, iar parser-ul folosește o gramatică specifică pentru a organiza aceste token-uri într-o structură logică, respectând regulile de sintaxă ale limbajului.

Pasul următor constă în definirea și implementarea comportamentului regulilor de parsare. Acest lucru se realizează prin intermediul funcțiilor Visit, care fac parte dintr-un vizitator (visitor pattern). Aceste funcții au scopul de a parcurge diferitele contexte generate de parser și de a aplica logica de procesare corespunzătoare fiecărui context.

Un context reprezintă o entitate logică derivată din gramatică, cum ar fi o expresie, o declarație, sau un bloc de cod. ANTLR generează clase pentru fiecare dintre aceste contexte, iar funcțiile Visit permit procesarea fiecărei entități în mod specific.

ANTLR generează o clasă de bază, precum BaseVisitor, din care poți deriva o clasă personalizată. Această clasă este utilizată pentru a suprascrie metodele care corespund fiecărui context din gramatică. Fiecare metodă Visit procesează un tip specific de context. Logica implementată în fiecare metodă Visit este personalizată pentru a interpreta și valida structura sintactică, precum și pentru a executa acțiuni asociate.

Clasele generate de ANTLR includ, pe lângă BaseVisitor, și alte componente precum BaseListener și clase contextuale pentru fiecare regulă din gramatică. Aceste contexte conțin structura sintactică a codului, fiind folosite pentru traversarea arborelui și procesarea fiecărui nod.

În plus, ANTLR permite extensibilitate prin implementarea vizitatorilor personalizați (Visitor Pattern) sau a ascultătorilor (Listener Pattern), oferind flexibilitate în analizarea, validarea și interpretarea limbajului. Prin combinarea gramaticilor bine definite și a logicii de procesare personalizate, ANTLR facilitează crearea de compilatoare, interpretoare și alte instrumente de procesare a limbajelor.

2.2 Exemplu: VisitIfStmt

```
ifStmt:
    IF_KEYWORD LPAREN expr RPAREN LBRACE statement+ RBRACE (
        elseStmt
    )?;
```

Figura 8: Regula de parsare pentru instrucțiunea if

```
public override object VisitIfStmt(BasicLanguageParser.IfStmtContext context)
{
    object conditionResult = EvaluateExpression(context.expr());
    if (conditionResult is bool condition)
    {
        if (condition)
        {
            foreach (var statement in context.statement())
            {
                Visit(statement);
            }
        }
        else if (context.elseStmt() != null)
        {
            return Visit(context.elseStmt());
        }
    }
    else
    {
        throw new Exception("Type Error: If condition must evaluate to a boolean value.");
    }

    return null;
}
```

Figura 9: Funcție Visit pentru instrucțiunea if

Metoda VisitIfStmt procesează o instrucțiune if din arborele de sintaxă. Mai întâi evaluează expresia condițională (context.expr()) și verifică dacă rezultatul este de tip boolean. Dacă condiția este adevărată, parcurge și execută instrucțiunile din blocul if. În caz contrar, dacă există un bloc else, îl vizitează. Dacă expresia condițională nu este booleană, aruncă o excepție de tip.

2.3 Relevanța utilizării scopurilor

Pentru a ilustra funcționalitățile legate de variabile locale și globale, precum și gestionarea funcțiilor cu scop unic, am creat clasa Scope. Aceasta oferă un mecanism pentru stocarea și manipularea variabilelor într-o ierarhie de contexte (scope-uri), facilitând accesul și actualizarea variabilelor în funcție de nivelul lor (local sau global). Fiecare obiect Scope are propriul set de variabile și un scop părinte opțional, ceea ce permite propagarea căutărilor pentru variabile nedeclarate în scopul curent. Astfel, această clasă gestionează declararea, atribuirea și validarea tipurilor variabilelor, oferind un model flexibil pentru a sprijini funcțiile și blocurile de cod cu scop propriu.

```
11 references
public class Scope
{
    private Dictionary<string, (string type, object value)> variables;
    private Scope parent;
    4 references
    public Scope(Scope parentTemp = null)
    {
        variables = new Dictionary<string, (string, object)>();
        parent = parentTemp;
    }
```

Figura 10: Clasa Scope

2.4 Clasa MyBasicLanguageVisitor

Clasa personalizată MyBasicLanguageVisitor are următorii membrii:

- scopeStack - Este un stack care gestionează ierarhia de scopuri (local și global). Permite menținerea unui context de variabile pentru funcții, blocuri și instrucțiuni. Fiecare funcție, bloc sau instrucțiune poate introduce un nou scop și îl elimină la ieșire.
- globalScope - Reprezintă scopul global, în care sunt declarate variabilele și funcțiile accesibile în întregul program. Este inițializat o singură dată și servește drept punct de referință în cazul variabilelor globale.
- functionDeclarations - Stochează declarațiile funcțiilor din program. Cheia dicționarului este numele funcției, iar valoarea conține: tipul de retur al funcției, lista de parametri ai funcției, fiecare reprezentat printr-un tuplu (type, name) care indică tipul și numele parametrului, corpul funcției (codul care urmează să fie executat), reprezentat de nodurile StatementContext din arborele de sintaxă generat de ANTLR.
- returnStack - Este o stivă care gestionează valorile de retur ale funcțiilor. Fiecare tuplu din stivă conține: un indicator boolean care specifică dacă o funcție a returnat o valoare, valoarea returnată de funcție.

```
3 references
public class MyBasicLanguageVisitor : BasicLanguageBaseVisitor<object>
{

    private Stack<Scope> scopeStack;
    private Scope globalScope;
    private Dictionary<string, (string returnType, List<(string type, string name)> parameters, BasicLanguageParser.StatementContext[] body)> functionDeclarations;
    private Stack<(bool isReturning, object value)> returnStack;
```

Figura 11: Clasa MyBasicLanguageVisitor

2.5 Exemplu: VisitFunctionDefinition

Funcția VisitFunctionDefinition procesează definiția unei funcții și o înregistrează în dicționarul functionDeclarations. Extrage numele funcției, tipul de retur și lista de parametri din contextul furnizat de parser. Lista de parametri este reprezentată ca o listă de tupluri (string, string), unde fiecare tuplu conține tipul și numele unui parametru. Funcția verifică dacă există parametri cu nume duplicate, aruncând o excepție în caz de duplicat. De asemenea, se verifică dacă o funcție cu același nume și parametri identici a fost deja definită, pentru a preveni redefinirea. Dacă verificările trec, funcția este adăugată în functionDeclarations, stocând tipul de retur, lista parametrilor și corpul funcției.

```
functionDefinition:
    type VARIABLE_NAME LPAREN parameterList? RPAREN LBRACE statement* RBRACE;

parameterList: parameter (COMMA parameter)*;
```

Figura 12: Regulile de parsare pentru definirea unei funcții

```
3 references
public override object VisitFunctionDefinition(BasicLanguageParser.FunctionDefinitionContext context)
{
    string functionName = context.VARIABLE_NAME().GetText();
    string returnType = context.type().GetText();
    List<(string, string)> parameterList = context.parameterList()?.parameter()
.Select(p => (p.type().GetText(), p.VARIABLE_NAME().GetText())).ToList() ?? new List<(string, string)>();

    var seenParameters = new HashSet<string>();
    foreach (var param in parameterList)
    {
        if (!seenParameters.Add(param.Item2))
        {
            throw new Exception($"Error: Duplicate parameter '{param.Item2}'");
        }
    }
    if (functionDeclarations.ContainsKey(functionName))
    {
        var existingFunction = functionDeclarations[functionName];
        var existingParameters = existingFunction.parameters;
        if (existingParameters.Count == parameterList.Count)
        {
            bool parametersMatch = true;
            for (int i = 0; i < parameterList.Count; i++)
            {
                if (existingParameters[i].type != parameterList[i].Item1)
                {
                    parametersMatch = false;
                    break;
                }
            }
            if (parametersMatch)
            {
                throw new Exception($"Error: Duplicate function definition for '{functionName}' with identical parameter types.");
            }
        }
    }
    functionDeclarations[functionName] = (returnType, parameterList, context.statement());
    return null;
}
```

Figura 13: Funcția Visit pentru definirea unei funcții

2.6 Exemplu: VisitFunctionCall

Funcția VisitFunctionCall procesează apelurile funcțiilor, verificând dacă funcția există, dacă numărul și tipurile argumentelor corespund parametrilor, și executând instrucțiunile funcției. Lista de argumente este evaluată prin apelul EvaluateExpression, iar parametrii funcției sunt obținuți din definiția funcției stocată în functionDeclarations. Suprapunerea parametrilor și argumentelor se face folosind metoda Zip, care creează perechi între fiecare argument și parametrul corespunzător. Pentru fiecare pereche (arg, param), se extrag tipul și numele parametrului, iar argumentul este verificat și adăugat în scopul curent cu AddVariable. Acest lucru permite utilizarea valorilor argumentelor ca variabile locale în timpul execuției funcției. După maparea parametrilor la argumente, funcția creează un nou scop local (pus pe scopeStack), execută instrucțiunile funcției, și apoi curăță scopul la final. Valoarea returnată este convertită în tipul de retur specificat al funcției, asigurând compatibilitatea tipurilor.

```
functionCall: VARIABLE_NAME LPAREN argumentList? RPAREN;
argumentList: expr (COMMA expr)*;
```

Figura 14: Regulile de parsare pentru apelarea unei funcții

```
public override object VisitFunctionCall(BasicLanguageParser.FunctionCallContext context)
{
    string functionName = context.VARIABLE_NAME().GetText();
    if (!functionDeclarations.ContainsKey(functionName))
    {
        throw new Exception($"Error: Function Definition for {functionName} does not exist.");
    }
    List<object> argumentList = context.argumentList()?.expr()
.Select(a => EvaluateExpression(a)).ToList() ?? new List<object>();
    var (returnType, parameterList, instructions) = functionDeclarations[functionName];
    if (argumentList.Count() != parameterList.Count())
    {
        throw new Exception($"Error: Argument List and Parameter List d not correspond.");
    }
    scopeStack.Push(new Scope(globalScope));
    foreach (var pair in argumentList.Zip(parameterList, (arg, param) => (arg, param)))
    {
        var (arg, param) = pair;
        string paramType = param.Item1;
        string paramName = param.Item2;
        scopeStack.Peek().AddVariable(paramName, paramType, arg);
    }

    returnStack.Push((false, null));

    foreach (var instruction in instructions)
    {
        Visit(instruction);
    }
    var (isReturning, returnValue) = returnStack.Pop();

    scopeStack.Pop();

    return Scope.Converter(returnType, returnValue);
}
```

Figura 15: Funcția Visit pentru apelarea unei funcții

2.7 Funcția Main

Funcția CallMain evidențiază rolul special al funcției main în program, fiind punctul de intrare principal, așa cum este specific în limbaje precum C/C++. Toate apelurile de funcții și execuția codului încep din funcția main, ceea ce o face o supra-funcție care coordonează întregul flux de execuție al programului. În implementare, se creează un scop nou, local, pornind de la globalScope, pentru a izola variabilele și funcțiile definite în main. Apoi, corpul funcției main este parcurs instrucțiune cu instrucțiune folosind metoda Visit. După terminarea execuției, returnările (gestionate prin returnStack) și scopul creat sunt curățate pentru a menține consistența programului. Astfel, funcția main asigură inițializarea, apelurile altor funcții și managementul general al execuției, subliniind importanța sa ca punct central al oricărui program.

```
1 reference
public void CallMain()
{
    scopeStack.Push(new Scope(globalScope));
    returnStack.Push((false, null));
    if (functionDeclarations.ContainsKey("main"))
    {
        foreach (var instruction in functionDeclarations["main"].body)
        {
            Visit(instruction);
        }
    }
    returnStack.Pop();
    scopeStack.Pop();
    return;
}
```

Figura 16: Tratarea funcției main

3 Concluzii

Dezvoltarea unui interpretator pentru C/C++ este un proces complex și provocator, care implică înțelegerea profundă a limbajului și implementarea funcționalităților acestuia într-un mod coerent. Procesul începe cu definirea unei gramatici formale, utilizând un instrument precum ANTLR, pentru a genera lexer-ul și parser-ul necesare recunoașterii structurii codului. După această etapă, se construiește un arbore de sintaxă abstractă (AST) care reprezintă ierarhic structura codului și facilitează procesarea sa ulterioară.

Cea mai dificilă parte a fost implementarea logicii funcțiilor, care în C/C++ joacă un rol central, fiind utilizate atât pentru modularitate, cât și pentru gestionarea eficientă a resurselor. A trebuit să asigurăm suport pentru apeluri de funcții, manipularea parametrilor și argumentelor, precum și gestionarea variabilelor locale și globale, încercând să replicăm comportamentul specific limbajului C++. De asemenea, a fost necesară implementarea mecanismelor de scope și stack pentru a gestiona corect execuția instrucțiunilor și funcțiilor.

Crearea acestui interpretator a fost un exercițiu riguros de proiectare și implementare, incluzând parcurgerea arborelui de sintaxă, gestionarea tipurilor de date și validarea regulilor de sintaxă. Prin încercarea de a imita cât mai fidel comportamentul limbajului C++, am reușit să înțelegem mai bine complexitatea acestuia și să dezvoltăm o soluție funcțională care să demonstreze principiile fundamentale ale procesării limbajelor de programare.

Bibliografie

[1] C. Xing, *How Interpreters Work: An Overlooked Topic in Undergraduate Computer Science Education*, Proc. In CCSC Southern Eastern Conference, 2 December 2009.

[2] R. Mak, *Writing Compilers and Interpreters: A Modern Software Engineering Approach Using Java*, Wiley, 3rd edition, 2009.

[3] Fan Wu, Hira Narang, Miguel Cabral, *Design and Implementation of an Interpreter Using Software Engineering Concepts*, (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 5, No. 7, 2014.

[4] Compiler Design, Available from: `https://slideplayer.com/slide/16389165/`, [Accessed 11.01.2025]