الجمهورية العربية السورية

جامعة دمشق

كلية الهندسة المعلوماتية

مشروع المترجمات

Flask and Jinja2 with HTML and CSS for Web application

عمل الطلبة :

سنا عبدو رحيم

ماسة محمد بسام درغام

أية ممدوح عبد الغني

أمينة عمر عمر

أية محمد الأحمد

اولاً : مقدمة :

Flask:

هو إطار عمل ويب خفيف (Microframework) مكتوب بلغة بايثون، يتيح بناء تطبيقات ويب ديناميكية وتفاعلية بسهولة. يعتمد على مكتبة Werkzeug لمعالجة الطلبات وJinja2 لتوليد الصفحات.

Jinja2 :

هو محرك قوالب (Template Engine) يُستخدم مع Flask لتوليد صفحات HTML ديناميكية باستخدام متغيرات وحلقات شرطية.

ثانياً : البنى الأساسية لبناء تطبيق ويب:

- Application Object:

  الكائن الأساسي (__name__)Flask الذي يدير إعدادات التطبيق.

- Routing System:

  ربط عنوان URL بدالة بايثون عبر.app.route@

- Request/Response:

  التعامل مع الطلبات القادمة من المستخدم وإرجاع الاستجابات.

- Templates (Jinja2):

  توليد صفحات HTML ديناميكية باستخدام متغيرات وتوجيهات.

- Static Files:

دعم ملفات CSS و JavaScript والصور.

- Blueprints:

تنظيم المشروع إلى وحدات صغيرة قابلة لإعادة الاستخدام .

ثالثاً: بعض البنى التي استعملناها:

- **Function**:

تعريف الدوال لمعالجة الطلبات.

- **Variables**:

تخزين البيانات وإرسالها إلى القوالب.

- **Array Declaration**:

استخدام القوائم في بايثون لتمثيل بيانات متعددة.

- **Object Declaration**:

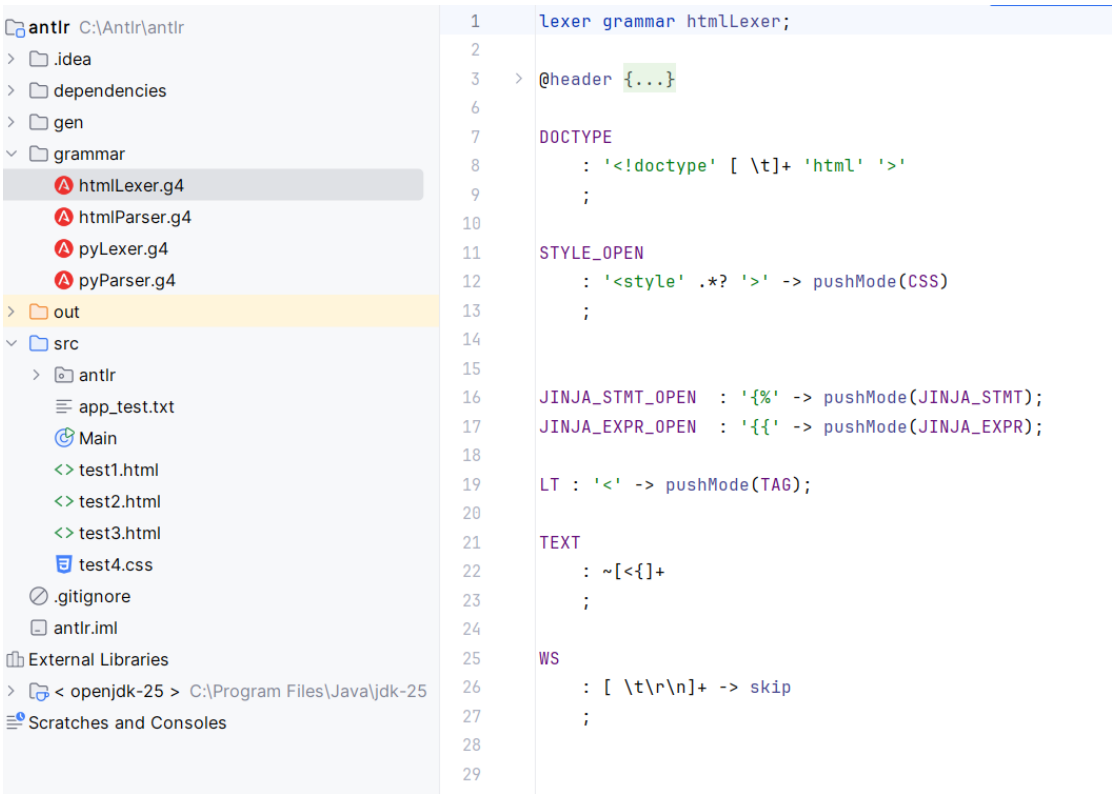
استخدام القواميس (dict) لتمثيل بيانات مركبة.

رابعاً :

بناء المترجم

تم تقسيم القواعد إلى :

1. بناء محلل لفظي flask
2. بناء محلل قواعد flask
3. بناء محلل لفظي HTML
4. بناء محلل قواعدHTML



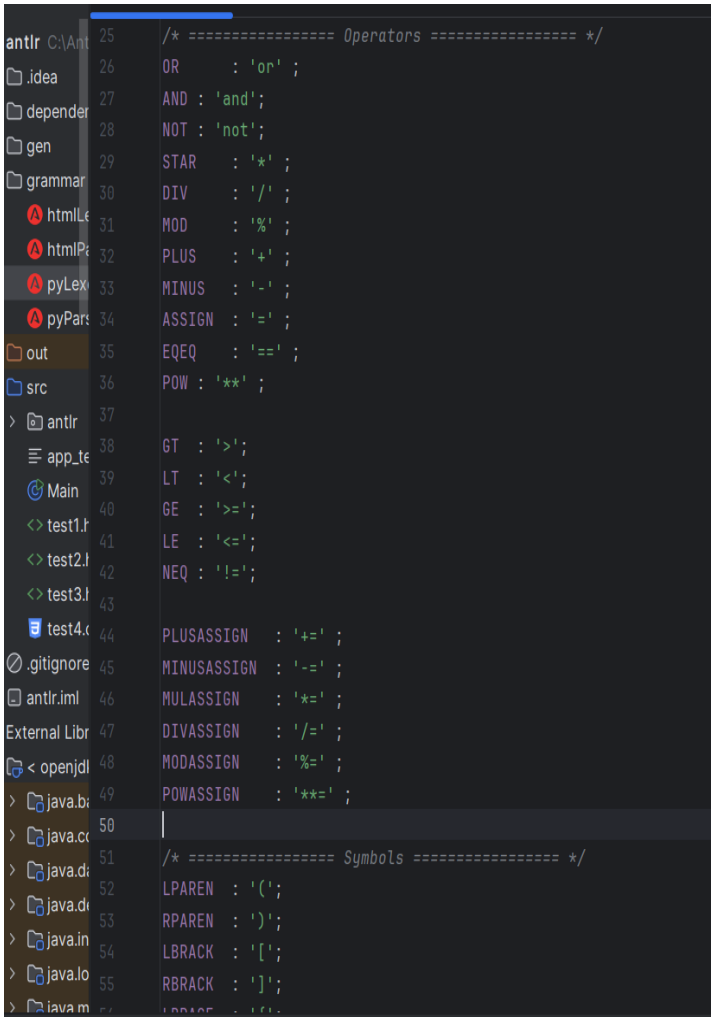
بناء المحلل اللفظي :

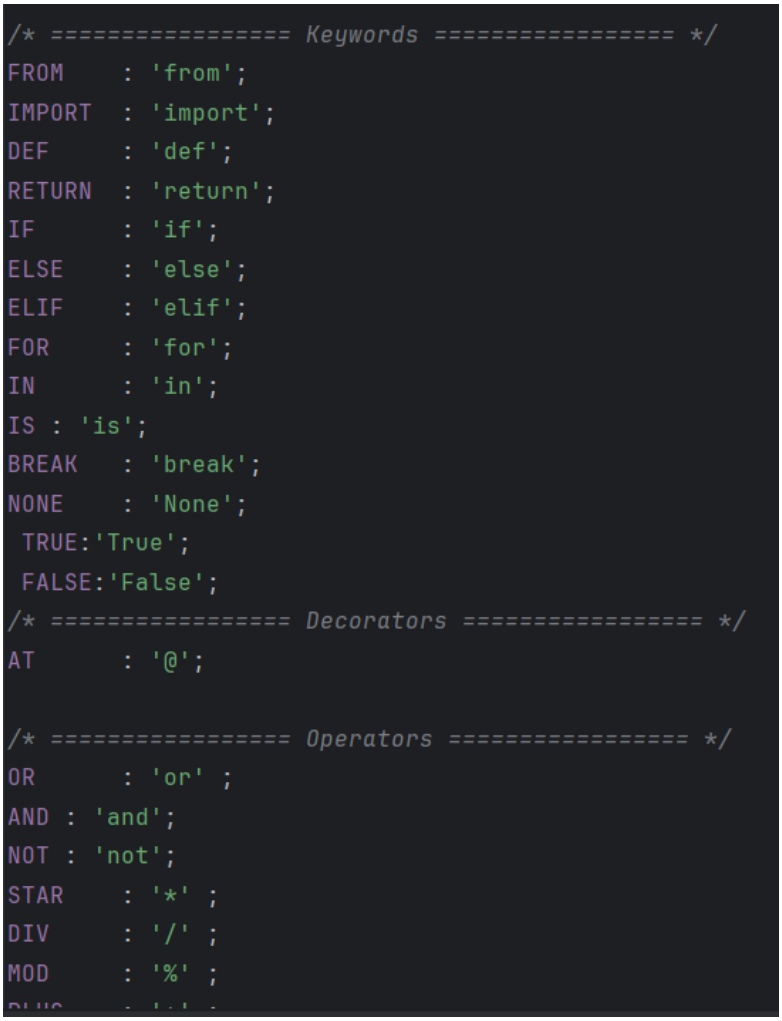
التحليل اللفظي هو المرحلة الأولى من مراحل بناء المترجم، حيث يتم قراءة الشيفرة المصدرية

من اليسار إلى اليمين وتحويلها إلى مجموعة من الرموز(Tokens)

في هذه المرحلة يتم تعريف الرموز الأساسية مثل:

- Identifiers
- Numbers
- Strings
- Keywords



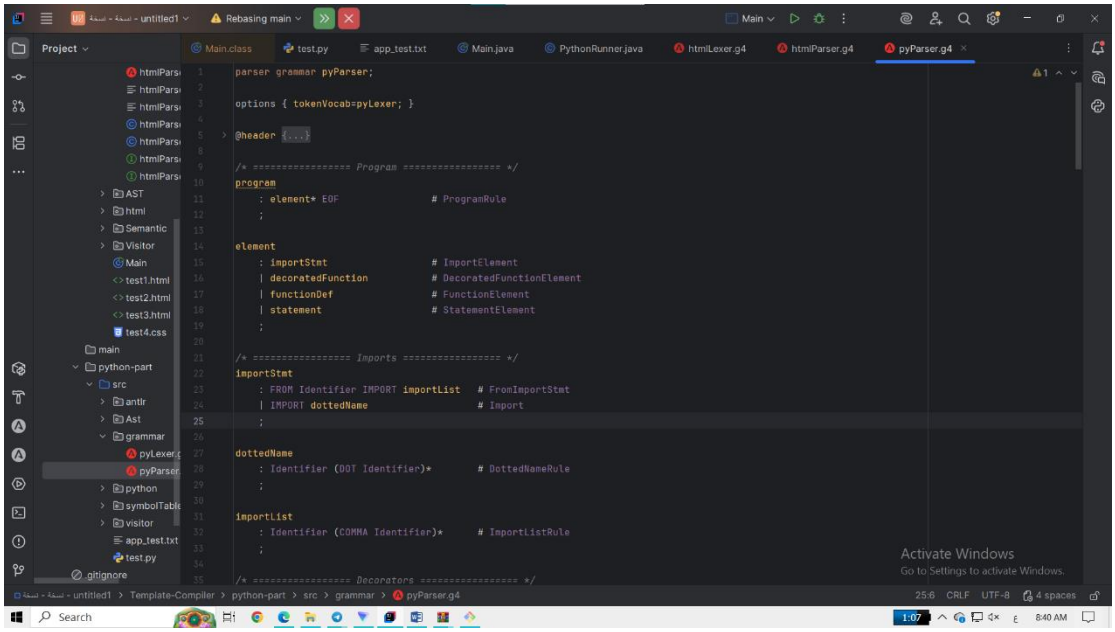


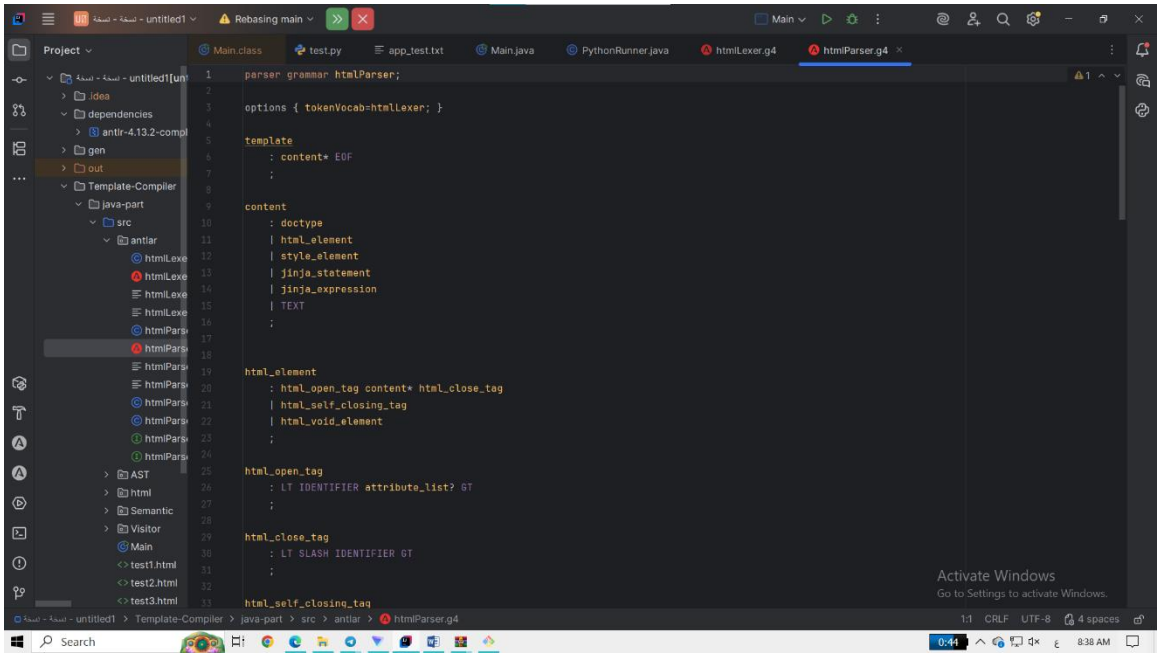
```
/* ================== Symbols ================== */
LPAREN   : '(';
RPAREN   : ')';
LBRACK   : '[';
RBRACK   : ']';
LBRACE   : '{';
RBRACE   : '}';
COMMA    : ',';
COLON    : ':';
DOT      : '.';

/* ================== Literals ================== */
StringLiteral
     : '"' (~["\\] | '\\' .)* '"'
     | '\'' (~['\\] | '\\' .)* '\''
     ;


NumberLiteral
     : [0-9]+
     ;

/* ================== Identifiers ================== */
Identifier
     : [a-zA-Z_][a-zA-Z0-9_]*
     ;

/* ================== Comments ================== */
COMMENT
```

بناء المحلل القواعدي (parser):

يُعد التحليل النحوي المرحلة الثانية من مراحل بناء المترجم، حيث يتم التحقق من صحة تركيب الشيفرة المصدرية بناءً على القواعد النحوية المحددة مسبقًا.

يقوم المحلل النحوي ببناء شجرة تحليل (Parse Tree)، ومن ثم يتم تحويلها إلى شجرة قواعد مجردة (AST) تُستخدم في المراحل اللاحقة.


parser grammar pyParser;
options { tokenVocab=pyLexer; }
@header {...}
/* ================ Program ================ */
program
: element* EOF # ProgramRule
;
element
: importStmt # ImportElement
| decoratedFunction # DecoratedFunctionElement
| functionDef # FunctionElement
| statement # StatementElement
;
/* ================ Imports ================ */
importStmt
: FROM Identifier IMPORT importList # FromImportStmt
| IMPORT dottedName # Import
;
dottedName
: Identifier (DOT Identifier)* # DottedNameRule
;
importList
: Identifier (COMMA Identifier)* # ImportListRule
;
/* ================ Decorators ================ */
Activate Windows
Go to Settings to activate Windows.


parser grammar htmlParser;
options { tokenVocab=htmlLexer; }
template
: content* EOF
;
content
: doctype
| html_element
| style_element
| jinja_statement
| jinja_expression
| TEXT
;
html_element
: html_open_tag content* html_close_tag
| html_self_closing_tag
| html_void_element
;
html_open_tag
: LT IDENTIFIER attribute_list? GT
;
html_close_tag
: LT SLASH IDENTIFIER GT
;
html_self_closing_tag
Activate Windows
Go to Settings to activate Windows.

بناء شجرة ال AST

Template Compiler (HTML & Jinja) – AST and Symbol Table :

تم إنشاء الشجرة المجرّدة (AST) لتمثيل بنية القالب بعد عملية التحليل النحوي. تعتمد الشجرة على كلاس أساسي هو ASTNode يحتوي على المعلومات العامة المشتركة بين جميع العقد، وأهمها رقم السطر. تم توريث هذا الكلاس في باقي الكلاسات لتمثيل مختلف عناصر اللغة مثل عناصرHTML ، النصوص، وتعليماتJinja .

تم تقسيم العقد إلى عقد محتوى (ContentNode) وعقد تعابير (ASTExpression) وعقد خاصة بتعليماتJinja ، وذلك لتحقيق تنظيم أوضح لبنية الشجرة. كما تم استخدام كلاس ASTTemplate كجذر للشجرة، حيث يحتوي على قائمة بالعقد التي تمثل كامل محتوى الملف.

بعض الكلاسات الموجودة كمثال:

1- تم تعريف الكلاس **ASTNode** ككلاس أساسي (Base Class) لجميع عقد الشجرة المجرّدة، حيث يحتوي على الخصائص المشتركة بين جميع العقد مثل رقم السطر في ملف الإدخال، بالإضافة إلى تعريف تابع مجرد لطباعة الشجرة.

```
package AST;

public abstract class ASTNode {  30 usages  AyaMohammadAlahmad *
    protected int line;

    public ASTNode(int line) {  5 usages  AyaMohammadAlahmad
        this.line = line;
    }


    public int getLine() { return line; }

    public abstract String print(String indent);  12 implementations  AyaMohammadAlahmad
}
```

2 - بعد الكلاس **ASTNode**، تم تعريف الكلاس **ContentNode** كطبقة وسيطة لتمثيل جميع العقد التي تشكل محتوى القالب.

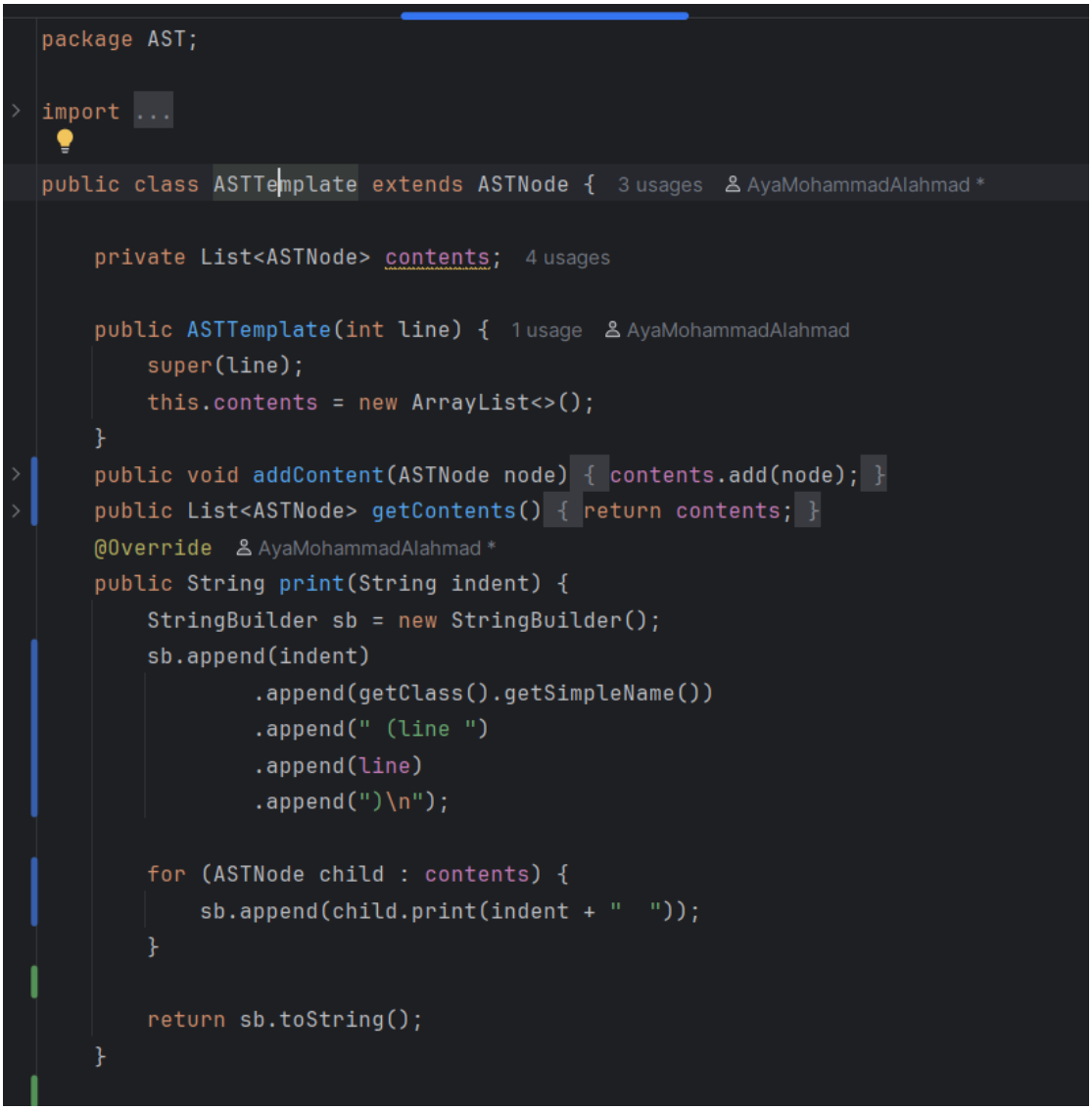
والذي يهدف إلى تنظيم بنية الشجرة وتمييز العقد القابلة للظهور داخل الصفحة مثل عناصر HTML، النصوص، وتعليماتJinja ، مما يسهل التعامل معها أثناء بناء الشجرة وطباعة محتواها.

3 - أيضاً تم تعريف الكلاس **ASTTemplate** الذي يُعتبر الجذر الأساسي للشجرة ، حيث يمثل الملف الكامل للقالب (Template) ويشكّل نقطة البداية التي تنطلق منها جميع العقد الأخرى.

يقوم هذا الكلاس بتجميع جميع عناصر القالب، سواء كانت عناصرHTML ، كتلJinja ، أو نصوص عادية، ضمن بنية واحدة منظمة، مما يسمح بالتعامل مع القالب كوحدة متكاملة أثناء مراحل التحليل اللاحقة.

يعتمد **ASTTemplate** على مبدأ التركيب(Composition) ، إذ يحتوي على قائمة من العقد الأبناء، بحيث تمثل هذه القائمة التسلسل الفعلي لمحتوى الملف كما ورد في القالب الأصلي. ويساهم هذا التنظيم في تسهيل عملية التنقل داخل الشجرة، سواء لأغراض الطباعة، التحليل الدلالي، أو بناء جدول الرموز.

حيث يرث من الكلاس الأساسي **ASTNode** ويقوم بإعادة تعريف دالة الطباعة، بما يضمن عرض بنية الشجرة بشكل هرمي واضح يعكس العلاقة بين العقد المختلفة.



```java
package AST;

import ...

public class ASTTemplate extends ASTNode {  3 usages  AyaMohammadAlahmad *

    private List<ASTNode> contents;  4 usages

    public ASTTemplate(int line) {  1 usage  AyaMohammadAlahmad
        super(line);
        this.contents = new ArrayList<>();
    }
    public void addContent(ASTNode node) { contents.add(node); }
    public List<ASTNode> getContents() { return contents; }
    @Override  AyaMohammadAlahmad *
    public String print(String indent) {
        StringBuilder sb = new StringBuilder();
        sb.append(indent)
                .append(getClass().getSimpleName())
                .append(" (line ")
                .append(line)
                .append(")\n");

        for (ASTNode child : contents) {
            sb.append(child.print(indent + "  "));
        }

        return sb.toString();
    }
```

4- كذلك تم تعريف الكلاس **HtmlElementNode** الذي يمثل عناصر لغة HTML ضمن الشجرة (AST)، حيث يُستخدم لتمثيل الوسوم المختلفة مثل html، head، body، div، وغيرها من العناصر الهيكلية.

يقوم هذا الكلاس بتخزين اسم الوسم(Tag Name) ، بالإضافة إلى قائمة السمات (Attributes)وقائمة الأبناء(Children) ، مما يسمح بتمثيل البنية الهرمية لعناصر HTML كما هي موجودة في القالب الأصلي. ويعكس هذا التصميم العلاقة الأبوية بين الوسوم، حيث يمكن لأي عنصر HTML أن يحتوي عناصر أخرى داخله.

يدعم **HtmlElementNode** كلًا من العناصر العادية والعناصر ذات الإغلاق الذاتي(Self-closing elements)، الأمر الذي يتيح تمثيل وسوم مثل Img و link و hr بطريقة صحيحة ضمن الشجرة.

حيث يرث من الكلاس **ContentNode**، ويقوم بإعادة تعريف دالة الطباعة لعرض اسم العقدة وسطرها في الملف، متبوعًا بعناصرها الأبناء بشكل هرمي، مما يساعد على فهم بنية المستند أثناء عملية التصحيح أو العرض.

يساهم **HtmlElementNode** في تسهيل مراحل التحليل اللاحقة، مثل التحليل الدلالي وبناء جدول الرموز، من خلال توفير تمثيل منظم وواضح لعناصر HTML داخل القالب.

```java
package AST;
import ...
public class HtmlElementNode extends ContentNode {  5 usages  AyaMohammadAlahmad *
    private String tagName;  1 usage
    private List<AttributeNode> attributes;  3 usages
    private List<ContentNode> children;  4 usages
    private boolean selfClosing;  1 usage
    public List<ContentNode> getChildren() {  1 usage  new *
        return children;}
    public HtmlElementNode(String tagName, boolean selfClosing, int line) {  3 usages  AyaMohammadAlahmad *
        super(line);
        this.tagName = tagName;
        this.selfClosing = selfClosing;
        this.attributes = new ArrayList<>();
        this.children = new ArrayList<>();}
    public void addAttribute(AttributeNode attr) { attributes.add(attr); }
    public void addChild(ContentNode node) { children.add(node); }
    @Override  AyaMohammadAlahmad *
    public String print(String indent) {
        StringBuilder sb = new StringBuilder();
        sb.append(indent)
                .append(getClass().getSimpleName())
                .append(" (line ")
                .append(line)
                .append(")\n");
        for (AttributeNode attr : attributes) {
            sb.append(attr.print(indent + "  "));}
        for (ContentNode child : children) {
            sb.append(child.print(indent + "  "));}
        return sb.toString();
    }
}
```

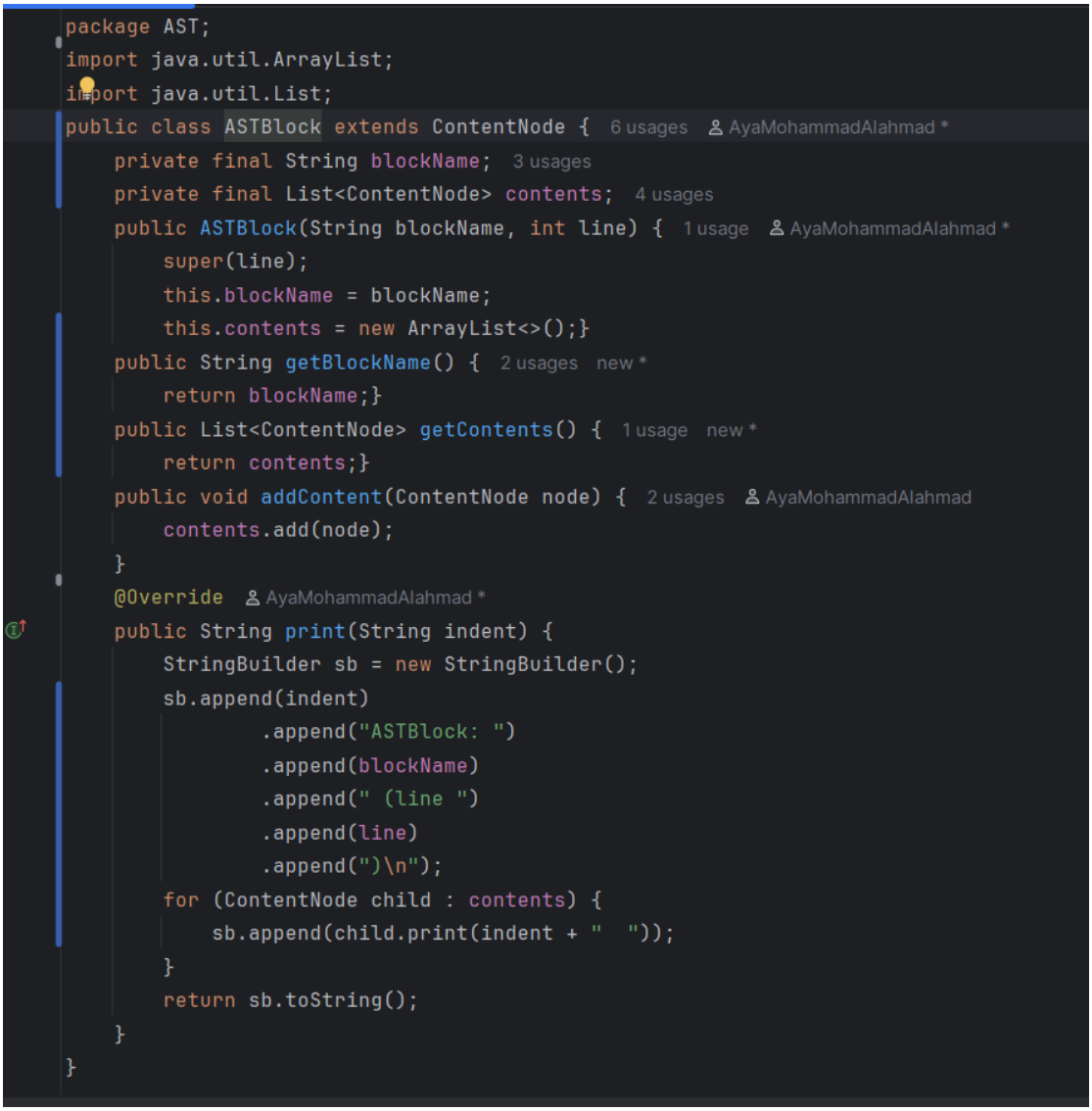
5- كذلك تم تعريف الكلاس **ASTBlock** الذي يمثل أوامر **Jinja block** ضمن الشجرة (AST)، حيث يُستخدم لتمثيل الكتل القابلة لإعادة التعريف أو الاستبدال داخل القالب.

يقوم هذا الكلاس بتخزين **اسم الـ block** (Block Name) بالإضافة إلى **رقم السطر** الذي تم فيه تعريفه، وقائمة بالعقد التابعة له (Children) التي تمثل المحتوى الموجود داخل الـblock ، سواء كان نصًا عاديًا، عناصرHTML ، أو تعابير Jinja أخرى.

يعكس تصميم **ASTBlock** البنية المنطقية للـ blocks في لغةJinja ، حيث يُعتبر الـ block وحدة مستقلة ذات معنى دلالي، وليس مجرد نص أو عنصر تركيبي. ويتم ربط جميع العناصر الواقعة بين {% block %}و {% endblock %}كأبناء لهذه العقدة، مما يسمح بتمثيل محتوى الـ block بشكل هرمي ومنظم داخل الشجرة.

يرث **ASTBlock** من الكلاس **ContentNode**، ويقوم بإعادة تعريف دالة الطباعة لعرض اسم الـ block ورقم السطر الخاص به، متبوعًا بمحتواه بشكل هرمي، الأمر الذي يسهل عملية تتبع بنية القالب وفهم تركيب الـ blocks أثناء التصحيح أو العرض.

يساهم **ASTBlock**في الربط بين التحليل النحوي والتحليل الدلالي، من خلال توفير تمثيل واضح ومنظم لبنية كتل Jinja داخل القالب.



```java
package AST;
import java.util.ArrayList;
import java.util.List;
public class ASTBlock extends ContentNode {  6 usages  AyaMohammadAlahmad *
    private final String blockName;  3 usages
    private final List<ContentNode> contents;  4 usages
    public ASTBlock(String blockName, int line) {  1 usage  AyaMohammadAlahmad *
        super(line);
        this.blockName = blockName;
        this.contents = new ArrayList<>();}
    public String getBlockName() {  2 usages  new *
        return blockName;}
    public List<ContentNode> getContents() {  1 usage  new *
        return contents;}
    public void addContent(ContentNode node) {  2 usages  AyaMohammadAlahmad
        contents.add(node);
    }
    @Override  AyaMohammadAlahmad *
    public String print(String indent) {
        StringBuilder sb = new StringBuilder();
        sb.append(indent)
                .append("ASTBlock: ")
                .append(blockName)
                .append(" (line ")
                .append(line)
                .append(")\n");
        for (ContentNode child : contents) {
            sb.append(child.print(indent + "  "));
        }
        return sb.toString();
    }
}
```

بعد الانتهاء من مرحلة التحليل النحوي وبناء الشجرة المجردة (AST) ، يتم تنفيذ مرحلة **طباعة الـ AST** بهدف عرض البنية الهرمية للقالب بشكل واضح ومنظم، مما يساعد على فهم ناتج التحليل والتحقق من صحة التمثيل الداخلي للبرنامج.

تعتمد آلية الطباعة في المشروع على مبدأ **الطباعة الهرمية (Hierarchical Printing)** ، حيث تقوم كل عقدة في الشجرة بطباعة نفسها أولًا، ثم تمرير عملية الطباعة إلى العقد الأبناء التابعة لها، مع زيادة مستوى الإزاحة (Indentation) في كل مستوى، بحيث يظهر تسلسل العلاقات الأبوية بين العقد بشكل بصري واضح.

تم تعريف دالة طباعة عامة في الكلاس الأساسي لعقد الـAST ، وتقوم كل عقدة متخصصة (مثل HtmlElementNodeو ASTBlockو TextNode) بإعادة تعريف هذه الدالة بما يتناسب مع طبيعتها ودورها داخل الشجرة. عند الطباعة، يتم عرض **نوع العقدة** ومعلوماتها الأساسية مثل (الاسم) أو اسم الـ (block) ورقم السطر في الملف الأصلي، مما يسهّل تتبع مصدر كل عنصر في القالب.

بالنسبة لعقد النصوص(TextNode) ، تتم طباعة محتوى النص مع رقم السطر فقط، دون وجود عقد أبناء. أما عقد عناصر HTML (HtmlElementNode) فتقوم بطباعة اسم الوسم ثم تمر على جميع العقد الأبناء التابعة لها (نصوص، عناصر HTML أخرى، أو تعابير (Jinja) وتقوم بطباعتها بشكل متداخل يعكس البنية الحقيقية للمستند.

أما عقد كتل Jinja (ASTBlock) فتتم طباعتها كعقد مستقلة تحمل اسم الـblock ، ثم يُطبع محتواها الداخلي بشكل هرمي، مما يوضح بوضوح حدود كل block ومحتواه داخل القالب.

تعتمد هذه الآلية على الاستدعاء الذاتي(Recursion) ، حيث تضمن أن يتم طباعة الشجرة كاملة ابتداءً من العقدة الجذرية (Template Node) وحتى أصغر عقدة نصية، مع الحفاظ على ترتيب العناصر كما ورد في الملف الأصلي.

تساهم هذه الطريقة في جعل ناتج الطباعة أداة فعّالة للتصحيح (Debugging) وفهم البنية الداخلية للقالب، كما تشكل أساسًا يمكن الاعتماد عليه في المراحل اللاحقة من المشروع مثل التحليل الدلالي وبناء جدول الرموز.

بعد الانتهاء من بناء الشجرة المجرّدة، تم اعتماد آلية موحّدة لطباعة الـ AST بهدف التحقق من صحة البنية الناتجة عن التحليل النحوي.

بناء Visitor

قمنا في البداية بتعريف كلاس **ASTVisitor** الذي يرث من htmlParserBaseVisitor<ASTNode>، حيث يقوم هذا الكلاس بزيارة عقد شجرة التحليل النحوي وتحويل كل عقدة منها إلى عقدة مناسبة في الشجرة المجرّدة(AST) ، مع تجاهل التفاصيل التركيبية غير الضرورية والاحتفاظ بالبنية الدلالية فقط.

- **آلية بناء الشجرة**

يبدأ بناء الشجرة من العقدة الجذرية عبر زيارة قاعدة **template**، حيث يتم إنشاء كائن من الكلاس **ASTTemplate** ليكون جذر الشجرة. بعد ذلك، تتم زيارة جميع عناصر المحتوى الموجودة في القالب وتحويلها إلى عقد AST مناسبة، ثم إضافتها إلى الجذر مع الحفاظ على ترتيبها الأصلي كما ورد في ملف الإدخال.

```java
package Visitor;

import AST.*;
import antlar.htmlParser;
import antlar.htmlParserBaseVisitor;

import java.util.Stack;

public class ASTVisitor extends htmlParserBaseVisitor<ASTNode> {  3 usages  AyaMohammadAlahmad *

    private final Stack<ASTBlock> blockStack = new Stack<>();  7 usages

    private ASTTemplate root;  4 usages

    @Override  1 usage  AyaMohammadAlahmad *
    public ASTNode visitTemplate(htmlParser.TemplateContext ctx) {
        root = new ASTTemplate(ctx.getStart().getLine());

        for (htmlParser.ContentContext c : ctx.content()) {
            ASTNode node = visit(c);
            if (node instanceof ContentNode) {
                root.addContent((ContentNode) node);
            }
        }
        return root;
    }
```

يعتمد الزائر على زيارة قاعدة **content** لتحديد نوع العنصر الحالي، سواء كان عنصر HTML، نصًا عاديًا، تعليمةJinja ، أو تعبيرJinja ، ومن ثم إنشاء العقدة المناسبة لكل حالة. يتم تمثيل النصوص بعقد من نوع **TextNode**، بينما يتم تمثيل عناصر HTML باستخدام **HtmlElementNode**، وتعليمات Jinja بعقد متخصصة.

```java
    @Override  1 usage  AyaMohammadAlahmad *
    public ASTNode visitContent(htmlParser.ContentContext ctx) {
        ASTNode node = null;

        if (ctx.doctype() != null) node = visit(ctx.doctype());
        else if (ctx.html_element() != null) node = visit(ctx.html_element());
        else if (ctx.style_element() != null) node = visit(ctx.style_element());
        else if (ctx.jinja_statement() != null) node = visit(ctx.jinja_statement());
        else if (ctx.jinja_expression() != null) node = visit(ctx.jinja_expression());
        else if (ctx.TEXT() != null)
            node = new TextNode(ctx.TEXT().getText(), ctx.getStart().getLine());
        if (node instanceof ContentNode && !blockStack.isEmpty()) {
            blockStack.peek().addContent((ContentNode) node);
            return null;
        }
        return node;
    }
```

- **التعامل مع عناصرHTML**

عند زيارة عناصرHTML ، يقوم الزائر بإنشاء عقدة من نوع **HtmlElementNode** تحتوي على اسم الوسم، رقم السطر، وقائمة السمات والأبناء. يتم التعامل مع العناصر العادية والعناصر ذات الإغلاق الذاتي (Self-closing elements) بشكل منفصل لضمان تمثيل صحيح لبنية المستند. كما يتم تحويل سمات HTML إلى عقد مستقلة من نوع **AttributeNode**وربطها بالعنصر الأب.

```
@Override 1 usage  AyaMohammadAlahmad *
public ASTNode visitHtml_element(htmlParser.Html_elementContext ctx) {
    HtmlElementNode node;
    if (ctx.html_open_tag() != null) {
        String tag = ctx.html_open_tag().IDENTIFIER().getText();
        node = new HtmlElementNode(tag, selfClosing: false, ctx.getStart().getLine());
        if (ctx.html_open_tag().attribute_list() != null) {
            for (var a : ctx.html_open_tag().attribute_list().attribute()) {
                ASTNode attr = visit(a);
                if (attr instanceof AttributeNode) {
                    node.addAttribute((AttributeNode) attr);
                }}}
        for (var c : ctx.content()) {
            ASTNode child = visit(c);
            if (child instanceof ContentNode) {
                node.addChild((ContentNode) child);
            }}return node;}
    if (ctx.html_self_closing_tag() != null) {
        String tag = ctx.html_self_closing_tag().IDENTIFIER().getText();
        return new HtmlElementNode(tag, selfClosing: true, ctx.getStart().getLine());}
    if (ctx.html_void_element() != null) {
        String tag = ctx.html_void_element().VOID_TAG().getText();
        return new HtmlElementNode(tag, selfClosing: true, ctx.getStart().getLine());}
    return null;
}
```

- **التعامل مع كتل(Blocks) Jinja**

تم اعتماد آلية خاصة لمعالجة كتل Jinja من نوع {% block %}و {% endblock %} باستخدام مكدس (Stack) من نوع **ASTBlock**. عند الوصول إلى تعليمة block ، يتم إنشاء عقدة جديدة من نوع **ASTBlock** ودفعها إلى المكدس، بينما عند الوصول إلى تعليمة endblockيتم إنهاء الكتلة وربطها بمحتواها الداخلي بشكل هرمي صحيح.

تسمح هذه الآلية بتمثيل البنية المتداخلة لكتل Jinja بدقة، وضمان ربط كل محتوى بالكتلة التي ينتمي إليها، سواء كان نصًا، عنصرHTML ، أو تعبير Jinja آخر.

- **تعابير Jinja وعناصر أخرى**

يتم تمثيل تعابير Jinja من النوع {{ }}باستخدام عقد من نوع **ASTJinjaExpression**، حيث يتم الاحتفاظ بنص التعبير ورقم السطر فقط دون الدخول في تفاصيل التنفيذ. كما يتم تمثيل عناصر التنسيق (Style) باستخدام عقد من نوع **ASTStyle**، والتي تخزن محتوى CSS المرتبط بها.

```java
@Override 1 usage new *
public ASTNode visitJinja_statement(htmlParser.Jinja_statementContext ctx) {
    String text = ctx.getText().trim();
    if (text.startsWith("{%") && text.endsWith("%}")) {
        text = text.substring(2, text.length() - 2).trim();}
    if (text.startsWith("block")) {
        String name = text.substring("block".length()).trim();
        ASTBlock block = new ASTBlock(name, ctx.getStart().getLine());
        blockStack.push(block);
        return null;}
    if (text.startsWith("endblock")) {
        if (!blockStack.isEmpty()) {
            ASTBlock finished = blockStack.pop();
            if (!blockStack.isEmpty()) {
                blockStack.peek().addContent(finished);
            } else {
                root.addContent(finished);
            }}
        return null;}
    return new ASTJinjaStatementNode(text, ctx.getStart().getLine());
}
@Override 1 usage new *
public ASTNode visitJinja_expression(htmlParser.Jinja_expressionContext ctx) {
    return new ASTJinjaExpression(ctx.getText(), ctx.getStart().getLine());
}
```

وبذلك شكّل ال **ASTVisitor** الحلقة الأساسية التي تربط بين ناتج التحليل النحوي والتمثيل الداخلي المجرد للقالب.

وبنفس الطريقة تم معالجة جميع قواعد ال Parser مع عمل Override لجميع التوابع.

Symbol table

تم في البداية تعريف الكلاس **SymbolTable** ليكون المسؤول عن تخزين وإدارة الرموز (Symbols)المستخرجة أثناء مرحلة التحليل الدلالي. يعتمد هذا الكلاس على بنية بيانات من نوع **Map** لربط أسماء الرموز بالمعلومات الخاصة بها، حيث يتم تخزين كل رمز باستخدام اسمه كمفتاح وقيمة من نوع **Symbol**تحتوي على نوع الرمز وموقعه في الملف.

يحتوي الكلاس SymbolTable على مرجع إلى **الأب (Parent Scope)** بالإضافة إلى قائمة من **الأبناء(Children Scopes)** ، مما يسمح بتمثيل (Nested Scopes) بشكل هرمي، كما هو مطلوب في لغات القوالب التي تدعم البُنى المتداخلة مثل كتل.Jinja

يوفر هذا الكلاس مجموعة من العمليات الأساسية، أهمها:

- **add**: لإضافة رمز جديد إلى جدول الرموز الحالي.

- **lookup:** للبحث عن رمز ضمن النطاق الحالي، وفي حال عدم وجوده يتم البحث في النطاقات الأب حتى الوصول إلى الجذر.
- **lookupLocal**: للبحث عن رمز ضمن النطاق الحالي فقط دون الرجوع إلى النطاقات الأعلى.
- **printHierarchy:** لطباعة جدول الرموز بشكل هرمي يوضح العلاقة بين النطاقات المختلفة.

```
import java.util.*;

public class SymbolTable {  13 usages  new *

    private Map<String, Symbol> symbols = new LinkedHashMap<>();  5 usages
    private SymbolTable parent;  5 usages
    private List<SymbolTable> children = new ArrayList<>();  2 usages
    public SymbolTable() {  1 usage  new *
        this.parent = null;}
    public SymbolTable(SymbolTable parent) {  1 usage  new *
        this.parent = parent;
        if (parent != null) {
            parent.children.add(this);}}
    public void add(Symbol symbol) {  4 usages  new *
        symbols.put(symbol.getName(), symbol);}
    public Symbol lookup(String name) {  5 usages  new *
        if (symbols.containsKey(name)) return symbols.get(name);
        if (parent != null) return parent.lookup(name);
        return null;}
    public Symbol lookupLocal(String name) {  no usages  new *
        return symbols.get(name);}
    public SymbolTable getParent() {  no usages  new *
        return parent;}
    public void printHierarchy(String indent) {  2 usages  new *
        for (Symbol s : symbols.values()) {
            System.out.println(indent + s);}
        for (SymbolTable child : children) {
            child.printHierarchy( indent: indent + "  ");}}
}
```

بعد تعريف بنية جدول الرموز، تم إنشاء الكلاس **SymbolTableVisitor** المسؤول عن تعبئة جدول الرموز من خلال المرور على الشجرة المجرّدة (AST) يقوم هذا الزائر بزيارة عقد الشجرة واحدة تلو الأخرى وتحليلها دلاليًا دون التأثير على بنية الشجرة نفسها.

يبدأ الزائر بإنشاء (Root Scope)، ثم ينتقل بشكل هرمي عبر عقد الشجرة. عند الوصول إلى عقد تمثل كتل(ASTBlock) Jinja ، يتم إنشاء نطاق جديد تابع للنطاق الحالي، مما يحقق مفهوم النطاقات المحلية. يتم تسجيل اسم كل block في جدول الرموز مع نوعه المناسب.

كما يقوم SymbolTableVisitor بمعالجة تعابير Jinja ، حيث يتم:

- تسجيل التوابع عند وجود استدعاء دالة.
- تسجيل المتغيرات المستخدمة داخل التعابير.

ويتم دائمًا التحقق من عدم وجود الرمز مسبقًا داخل نفس النطاق قبل إضافته.

يعتمد هذا الزائر على الربط بين بنية الـ AST وبنية جدول الرموز، مما يضمن تمثيلًا دلاليًا صحيحًا.

```
public class SymbolTableVisitor {  3 usages  new *
    private SymbolTable rootScope = new SymbolTable();  2 usages
    private SymbolTable currentScope = rootScope;  11 usages
    public SymbolTable getSymbolTable() {  1 usage  new *
        return rootScope;
    }
    public void visit(ASTNode node) {  4 usages  new *
        if (node == null) return;
        if (node instanceof ASTTemplate template) {
            for (ASTNode c : template.getContents()) {
                visit(c);
            }
        }
        else if (node instanceof HtmlElementNode element) {
            for (ContentNode c : element.getChildren()) {
                visit(c);
            }
        }
        else if (node instanceof ASTBlock block) {
            handleBlock(block);
        }
        else if (node instanceof ASTJinjaExpression expr) {
            handleJinjaExpression(expr);
        }
        else if (node instanceof ASTJinjaStatementNode stmtNode) {
            handleJinjaStatementNode(stmtNode);
        }
    }
    private void handleBlock(ASTBlock block) {  1 usage  new *
        if (currentScope.lookup(block.getBlockName()) == null) {
            currentScope.add(new Symbol(block.getBlockName(), SymbolKind.BLOCK, block.getLine()));
```

```
    }
    private void handleBlock(ASTBlock block) {  1 usage  new *
        if (currentScope.lookup(block.getBlockName()) == null) {
            currentScope.add(new Symbol(block.getBlockName(), SymbolKind.BLOCK, block.getL
        }
        SymbolTable previousScope = currentScope;
        currentScope = new SymbolTable(previousScope);

        for (ContentNode c : block.getContents()) {
            visit(c);
        }
        currentScope = previousScope;
    }
    private void handleJinjaExpression(ASTJinjaExpression expr) {  1 usage  new *
        String text = expr.getText().trim();
        if (text.contains("(")) {
            String name = text.substring(0, text.indexOf("("));
            if (currentScope.lookup(name) == null) {
                currentScope.add(new Symbol(name, SymbolKind.FUNCTION, expr.getLine()));
            }
        } else if (!text.isEmpty()) {
            if (currentScope.lookup(text) == null) {
                currentScope.add(new Symbol(text, SymbolKind.VARIABLE, expr.getLine()));
            }
        }
    }
    private void handleJinjaStatementNode(ASTJinjaStatementNode stmt) {  1 usage  new *
        String text = stmt.getText().trim();
        if (!text.startsWith("endblock") && !text.isEmpty()) {
            if (currentScope.lookup(text) == null) {
                currentScope.add(new Symbol(text, SymbolKind.VARIABLE, stmt.getLine()));
```

بناء شجرة ال AST

Flask\ python : Ast and SymbolTable

تم إنشاء Package خاص باسمAST

ويحتوي على مجموعة من الكلاسات التي تمثل كل عناصر اللغة.

Element:

هو الكلاس الأب لكل عقد الـ AST

يحتوي على: اسم العقدة رقم السطر

يفرض على جميع العقد تنفيذ دالة print

الهدف منه:

1. توحيد جميع أنواع العقد
2. تسهيل التعامل معها داخل الـ Visitors

```java
package Ast;

public abstract class Element {

    protected String nodeName;
    protected int lineNumber;

    public Element(String nodeName, int lineNumber) {
        this.nodeName = nodeName;
        this.lineNumber = lineNumber;
    }

    protected void printIndent(int indent) {
        for (int i = 0; i < indent; i++) {
            System.out.print("  ");
        }
    }

    // كل عنصر يطبع نفسه
    public abstract void print(int indent);  25 implementations
}
```

Program Node

يمثل البرنامج كاملًا، ويحتوي على:

قائمة من العناصر (Imports، Functions، Statements)

هذه العقدة هي جذر الشجرة (Root).

```java
package Ast;

import ...

public class Program extends Element {  2 usages

    private List<Element> elements;  3 usages

    public Program(int lineNumber) {  2 usages
        super( nodeName: "Program", lineNumber);
        elements = new ArrayList<>();
    }

    public void addElement(Element element) { elements.add(element); }

    @Override
    public void print(int indent) {

        printIndent(indent);
        System.out.println(nodeName + " (line " + lineNumber + ")");

        for (Element element : elements) {
            element.print(indent + 1);
        }
    }
}
```

ImportElement

يمثل أوامر الاستيراد مثل:

import flask from flask import render_template

ويحتوي على:

- اسم الموديول
- قائمة الأسماء المستوردة

```java
package Ast;

import java.util.List;

public class ImportElement extends Element {  2 usages

    private String module;  3 usages
    private List<String> names;  4 usages

    public ImportElement(String module, List<String> names, int lineNumber) {  2 usages
        super( nodeName: "ImportElement", lineNumber);
        this.module = module;
        this.names = names;
    }

    @Override
    public void print(int indent) {

        printIndent(indent);
        System.out.println(nodeName + " (line " + lineNumber + ")");

        printIndent(indent + 1);
        if (names == null || names.isEmpty()) {
            System.out.println("module = " + module);
        } else {
            System.out.println("module = " + module + ", names = " + names);
        }
    }
}
```

FunctionDef

يمثل تعريف الدالة ويحتوي على:

- اسم الدالة
- المعاملات (Parameters)
- جسم الدالة (Block)

```java
public class FunctionDef extends Element {  5 usages

    private String name;           // اسم الدالة  2 usages
    private List<String> parameters; // أسماء المعاملات  2 usages
    private Block body;            // جسم الدالة  2 usages

    public FunctionDef(String name,  1 usage
                       List<String> parameters,
                       Block body,
                       int lineNumber) {
        super( nodeName: "FunctionDef", lineNumber);
        this.name = name;
        this.parameters = parameters;
        this.body = body;
    }

    @Override
    public void print(int indent) {
        printIndent(indent);
        System.out.println(nodeName + " (line " + lineNumber + ") : " + name);

        printIndent(indent + 1);
        System.out.println("Parameters: " + parameters);

        printIndent(indent + 1);
        System.out.println("Body:");
        body.print(indent + 2);
    }
}
```

DecoratedFunction

يمثل دالة مرفقة بـ decorators مثل Flask routes.

يتكون من:

1. قائمة decorators (تمثل كـ Call Expressions)
2. تعريف الدالة نفسها

هذا التصميم يعكس البنية الحقيقية لـ Flask.

```java
public class DecoratedFunction extends Element {  1 usage

    private List<CallExpr> decorators; // decorators  2 usages
    private FunctionDef function;      // الدالة نفسها  2 usages

    public DecoratedFunction(List<CallExpr> decorators,  1 usage
                             FunctionDef function,
                             int lineNumber) {
        super( nodeName: "DecoratedFunction", lineNumber);
        this.decorators = decorators;
        this.function = function;
    }

    @Override
    public void print(int indent) {
        printIndent(indent);
        System.out.println(nodeName + " (line " + lineNumber + ")");

        printIndent(indent + 1);
        System.out.println("Decorators:");
        for (CallExpr d : decorators) {
            d.print(indent + 2);
        }

        printIndent(indent + 1);
        System.out.println("Function:");
        function.print(indent + 2);
    }
}
```

Block

يمثل مجموعة من الأوامر المتتالية داخل:

- دالة
- شرط
- حلقة

ويحتوي على قائمة من Element.

```java
package Ast;

import java.util.List;

public class Block extends Element{  24 usages
    private List<Element>statements;  3 usages

    public Block(String nodeName, int lineNumber, List<Element> statements) {
        super( nodeName: "Block", lineNumber);
        this.statements = statements;
    }
    public List<Element> getStatements() {  1 usage
        return statements;
    }

    @Override
    public void print(int indent) {
        printIndent(indent);
        System.out.println(nodeName + " (line " + lineNumber + ")");
        for (Element s : statements) {
            s.print(indent + 1);
        }
    }
}
```

Statements Nodes

تم تمثيل كل Statement بكلاس مستقل:

1. AssignmentStmt ← إسناد قيمة لمتغير
2. ReturnStmt ← إرجاع قيم
3. IfStmt ← شرط if / else
4. ForStmt ← حلقة for
5. BreakStmt ← كسر الحلقة

Expressions Nodes

التعبيرات هي جوهر أي لغة، وتم تصميمها بشكل هرمي:

Expression (كلاس مجرد)

BinaryExpr ← العمليات الثنائية (+, ==, etc.)

UnaryExpr ← العمليات الأحادية (-x)

CallExpr ← استدعاء دالة

ConditionalExpr ← التعبير الشرطي

LiteralExpr ← القيم الثابتة

ListExpr / DictExpr ← الهياكل

IndexExpr ← الوصول للعناصر

ListComprehensionExpr → List comprehension

هذا التصميم يحافظ على أولوية العمليات (Operator Precedence).

```java
package Ast;

public abstract class Expression extends Element {
    public Expression(String nodeName, int lineNumber) { super(nodeName, lineNumber); }

    // كل Expression يجب أن يعرف كيف يطبع نفسه
    @Override  14 implementations
    public abstract void print(int indent);
}
```

Arguments

تم دعم:

- Positional arguments
- Named arguments
- Generator arguments

وذلك لتغطية استدعاءات الدوال في Python و Flask.

```java
package Ast;

public abstract class ArgumentExp extends Expression{  2 usa

    public ArgumentExp(String nodeName, int lineNumber) {
        super(nodeName, lineNumber);
    }
}
```

مرحلة (AstBuilder) Visitor:

AstBuilder

- وظيفته الأساسية:

تحويل Parse Tree الناتجة من ANTLR إلى AST مخصص

- آلية العمل:

لكل Rule في الـ Grammar يوجد visit و كل visit يزور أبناء العقدة ينشئ Node من نوع AST مناسب يعيد العقدة للأب

مثال:

Assignment Rule → AssignmentStmt

Expression Rule → BinaryExpr أو LiteralExpr

1. نقوم بإنشاء كائن Program.
2. لكل عنصر (element) في البرنامج، نستدعي visit لبناء AST فرعي.
3. نجمع كل العناصر في Program.

```java
import Ast.*;
import antlr.pyParser;
import antlr.pyParserBaseVisitor;
import org.antlr.v4.runtime.tree.ParseTree;
import org.antlr.v4.runtime.tree.TerminalNode;

import java.util.ArrayList;
import java.util.List;

public class AstBuilder extends pyParserBaseVisitor<Element> {  3 usages

    // ================= Program =================
    @Override  1 usage
    public Element visitProgramRule(pyParser.ProgramRuleContext ctx) {
        Program program = new Program(ctx.start.getLine());
        for (pyParser.ElementContext ectx : ctx.element()) {
            Element el = visit(ectx);
            if (el != null) program.addElement(el);
        }
        return program;
    }
```

التعابير Expressions:

- بناء التعبيرات

تم بناء التعبيرات بشكل متدرج:

Additive

Multiplicative

Comparison

Conditional

وهذا يمنع أخطاء الأسبقية في العمليات الحسابية والمنطقية. كل قاعدة لغوية تُحوّل إلى كلاس مناسب في AST:

نستخدم Recursive Descent لبناء تعابير ثنائية BinaryExpr أو أحادية UnaryExpr.

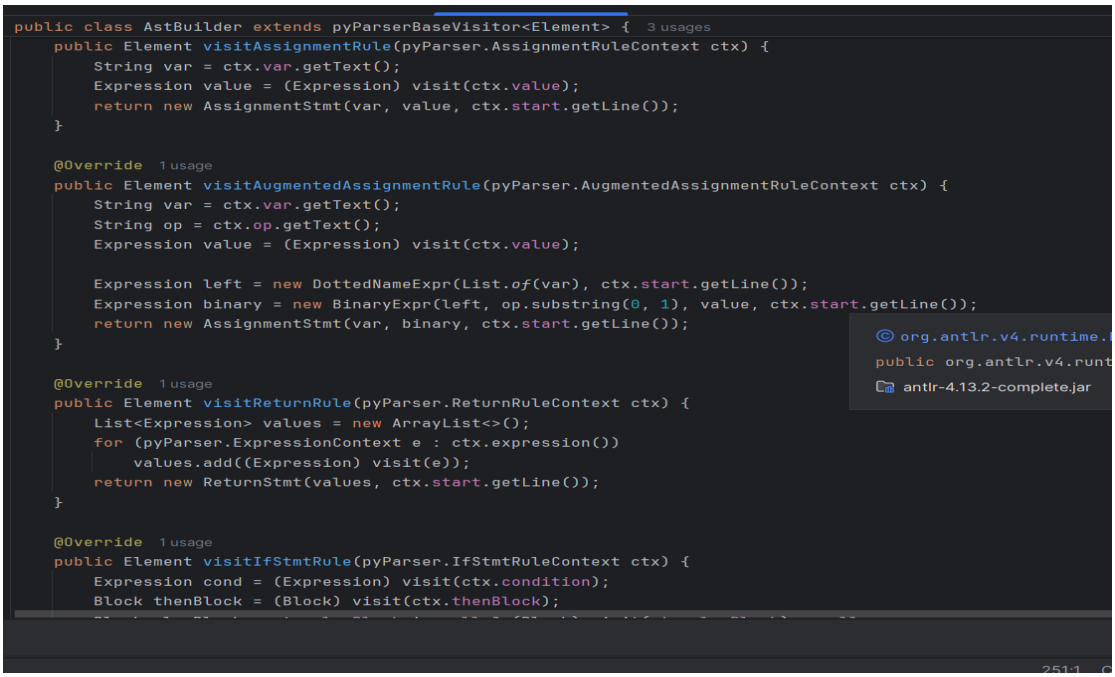
يتم تخزين رقم السطر لسهولة تتبع الأخطاء.

```java
public class AstBuilder extends pyParserBaseVisitor<Element> {  3 usages
    // ================= AdditiveExprRule =================
    @Override public Element visitAdditiveExprRule(pyParser.AdditiveExprRuleContext ctx) {  1 usage
        Expression result = (Expression) visit(ctx.multiplicativeExpr( i: 0));
        for (int i = 1; i < ctx.multiplicativeExpr().size(); i++) {
            String op = ctx.getChild( i: 2 * i - 1).getText();
            Expression right = (Expression) visit(ctx.multiplicativeExpr(i));
            result = new BinaryExpr(result, op, right, ctx.start.getLine());
        }
        return result;
    }

    // ================= MultiplicativeExprRule =================
    @Override public Element visitMultiplicativeExprRule(pyParser.MultiplicativeExprRuleContext ctx) {  1 usage
        Expression result = (Expression) visit(ctx.powerExpr( i: 0));
        for (int i = 1; i < ctx.powerExpr().size(); i++) {
            String op = ctx.getChild( i: 2 * i - 1).getText();
            Expression right = (Expression) visit(ctx.powerExpr(i));
            result = new BinaryExpr(result, op, right, ctx.start.getLine());
        }
        return result;
    }

    // ================= PowerExprRule =================
    @Override public Element visitPowerExprRule(pyParser.PowerExprRuleContext ctx) {  1 usage
        Expression left = (Expression) visit(ctx.unaryExpr( i: 0));
        if (ctx.unaryExpr().size() == 1) return left;
        Expression right = (Expression) visit(ctx.unaryExpr( i: 1));
        return new BinaryExpr(left,  operator: "**", right, ctx.start.getLine());
    }
```

- نبدأ بزيارة كل Decorator أولاً.
- ثم زيارة جسم الدالة.
- إنشاء DecoratedFunction مع جميع المعلومات.

```java
public class AstBuilder extends pyParserBaseVisitor<Element> {  3 usages
        return visit(ctx.functionDef());
    }
    @Override  1 usage
    public Element visitDecoratedFunctionRule(pyParser.DecoratedFunctionRuleContext ctx) {
        // زيارة كل decorator في القائمة 1
        List<CallExpr> decorators = new ArrayList<>();
        if (ctx.decorator() != null) {
            for (pyParser.DecoratorContext decCtx : ctx.decorator()) {
                decorators.add((CallExpr) visit(decCtx));
            }
        }

        // زيارة الدالة نفسها 2
        FunctionDef function = (FunctionDef) visit(ctx.functionDef());

        // DecoratedFunction AST node إنشاء 3
        return new DecoratedFunction(decorators, function, ctx.start.getLine());
    }

    @Override  1 usage
    public Element visitFunctionDefRule(pyParser.FunctionDefRuleContext ctx) {
        String name = ctx.name.getText();

        List<String> params = new ArrayList<>();
        if (ctx.parameterList() != null) {
            for (TerminalNode id : ctx.parameterList().getTokens(pyParser.Identifier))
                params.add(id.getText());
        }
```

- زيارة Statements



```
public class AstBuilder extends pyParserBaseVisitor<Element> {  3 usages
    public Element visitAssignmentRule(pyParser.AssignmentRuleContext ctx) {
        String var = ctx.var.getText();
        Expression value = (Expression) visit(ctx.value);
        return new AssignmentStmt(var, value, ctx.start.getLine());
    }

    @Override  1 usage
    public Element visitAugmentedAssignmentRule(pyParser.AugmentedAssignmentRuleContext ctx) {
        String var = ctx.var.getText();
        String op = ctx.op.getText();
        Expression value = (Expression) visit(ctx.value);

        Expression left = new DottedNameExpr(List.of(var), ctx.start.getLine());
        Expression binary = new BinaryExpr(left, op.substring(0, 1), value, ctx.start.getLine());
        return new AssignmentStmt(var, binary, ctx.start.getLine());
    }

    @Override  1 usage
    public Element visitReturnRule(pyParser.ReturnRuleContext ctx) {
        List<Expression> values = new ArrayList<>();
        for (pyParser.ExpressionContext e : ctx.expression())
            values.add((Expression) visit(e));
        return new ReturnStmt(values, ctx.start.getLine());
    }

    @Override  1 usage
    public Element visitIfStmtRule(pyParser.IfStmtRuleContext ctx) {
        Expression cond = (Expression) visit(ctx.condition);
        Block thenBlock = (Block) visit(ctx.thenBlock);
```

org.antlr.v4.runtime.
public org.antlr.v4.runt
antlr-4.13.2-complete.jar

Symbol table

في أي برنامج مترجم أو مفسر للغة برمجة، يحتاج النظام إلى تتبع جميع الرموز (Symbols) المستخدمة في البرنامج مثل المتغيرات، الدوال، والمعاملات.

الـ Symbol Table هو هيكل بيانات أساسي يقوم بتخزين هذه الرموز مع معلوماتها المرتبطة، مثل:

- الاسم
- النوع
- القيمة الحالية
- نطاق الصلاحية (Scope)

يُعتبر هذا الهيكل جزءًا من مرحلة التحليل الدلالي (Semantic Analysis) بعد التحليل النحوي (Parsing) في عملية الترجمة.

```java
package symbolTable;

public class VariableSymbol extends Symbol {  10 usages  ayamohamdabd

    private Object value;  4 usages
    private String internalName;  3 usages

    public VariableSymbol(String name, String type, Object value, String internalName)
        super(name, type);
        this.value = value;
        this.internalName = internalName;
    }

    public Object getValue() {  ayamohamdabd
        return value;
    }

    public void setValue(Object value) {  ayamohamdabd
        this.value = value;
    }

    public String getInternalName() {  no usages  ayamohamdabd
        return internalName;
    }

    @Override  ayamohamdabd
    public String toString() {
        return name + " : " + type + " = " + value + " (" + internalName + ")";
    }
}
```

- Scope

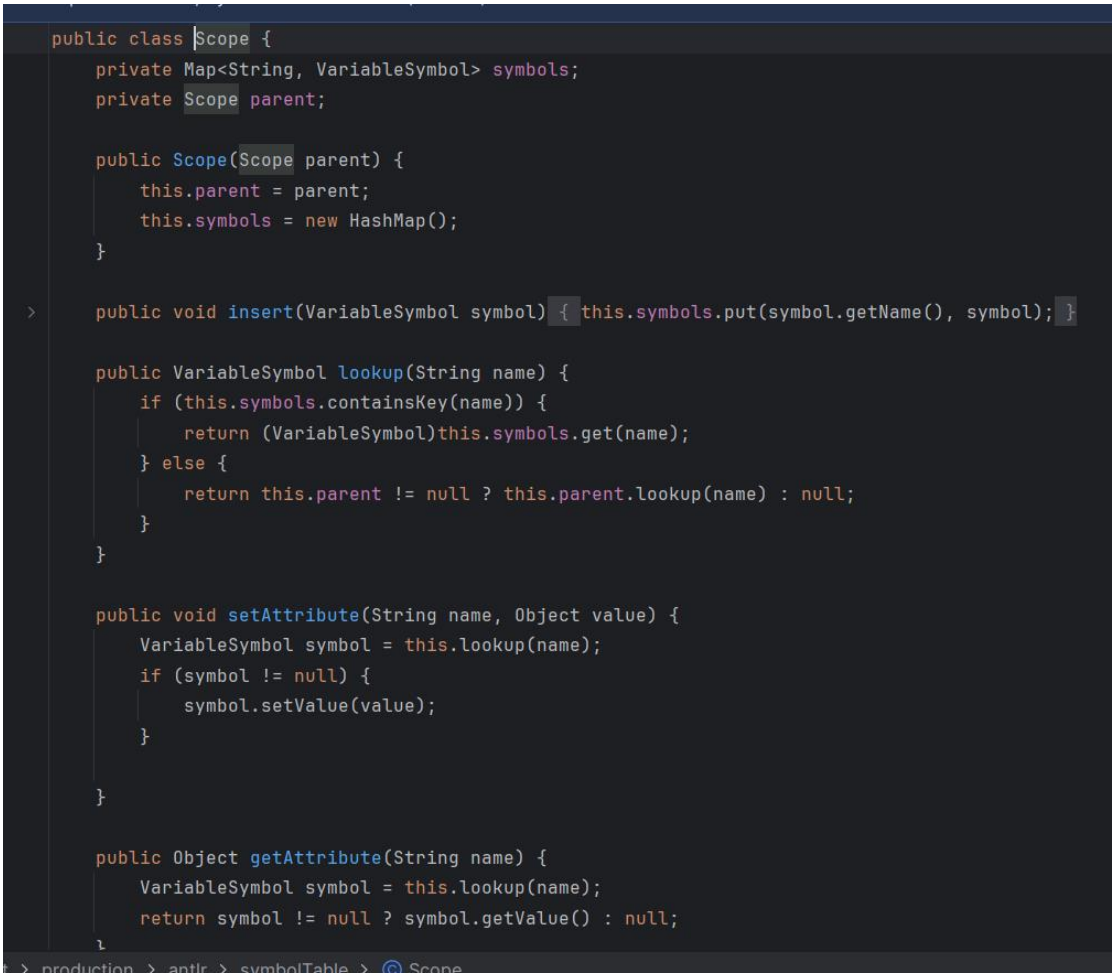
يمثل نطاقًا (Scope) في البرنامج مثل global scope أو scope داخل دالة أو حلقة.

- الخصائص الرئيسية:

1. symbols: خريطة تربط أسماء الرموز بالـ VariableSymbol.
2. parentScope: للإشارة إلى النطاق الخارجي، لتسهيل البحث عن الرموز الصاعدة

- الوظائف:

1. إضافة رمز جديد (insert).
2. البحث عن رمز موجود (lookup).



```
public class Scope {
    private Map<String, VariableSymbol> symbols;
    private Scope parent;

    public Scope(Scope parent) {
        this.parent = parent;
        this.symbols = new HashMap();
    }

    public void insert(VariableSymbol symbol) { this.symbols.put(symbol.getName(), symbol); }

    public VariableSymbol lookup(String name) {
        if (this.symbols.containsKey(name)) {
            return (VariableSymbol)this.symbols.get(name);
        } else {
            return this.parent != null ? this.parent.lookup(name) : null;
        }
    }

    public void setAttribute(String name, Object value) {
        VariableSymbol symbol = this.lookup(name);
        if (symbol != null) {
            symbol.setValue(value);
        }

    }

    public Object getAttribute(String name) {
        VariableSymbol symbol = this.lookup(name);
        return symbol != null ? symbol.getValue() : null;
    }
```

SymbolTable:

يمثل الجدول الرئيسي لإدارة الرموز.

- الخصائص:

currentScope: يشير إلى النطاق الحالي أثناء تحليل البرنامج

الوظائف:

- insert(symbol): إدخال رمز جديد في الـ current scope.
- lookup(name): البحث عن رمز في النطاق الحالي أو الصعود للـ parent scopes.
- setAttribute و getAttribute: تحديث واسترجاع القيم.
- allocateScope() و freeScope(): إنشاء نطاق جديد عند دخول دالة أو block، وتحريره عند الخروج.

```java
package symbolTable;

public class SymbolTable {  3 usages  ayamohamdabd

    private Scope currentScope;  10 usages
    private int memoryCounter = 0;  1 usage

    public SymbolTable() {  1 usage  ayamohamdabd
        allocate(); // Global scope
    }

    // allocate
    public void allocate() {  4 usages  ayamohamdabd
        currentScope = new Scope(currentScope);
    }

    // free
    public void free() {  3 usages  ayamohamdabd
        if (currentScope != null) {
            currentScope = currentScope.getParent();
        }
    }

    // insert
    public void insert(String name, String type, Object value) {  4 usages  ayamoh
        String internalName = "_var" + memoryCounter++;
        VariableSymbol symbol =
                new VariableSymbol(name, type, value, internalName);
        currentScope.insert(symbol);
    }

    // lookup
```

SymbolTableVisitor

زائر AST (Abstract Syntax Tree) لبناء Symbol Table.

- المهام الرئيسية:

1. زيارة العقد المختلفة في الكود.
2. إدخال المتغيرات الجديدة في الـ Symbol Table.
3. إنشاء Scopes جديدة عند دخول Functions, Blocks, Loops, If Statements
4. تحديث القيم للمتغيرات الموجودةز

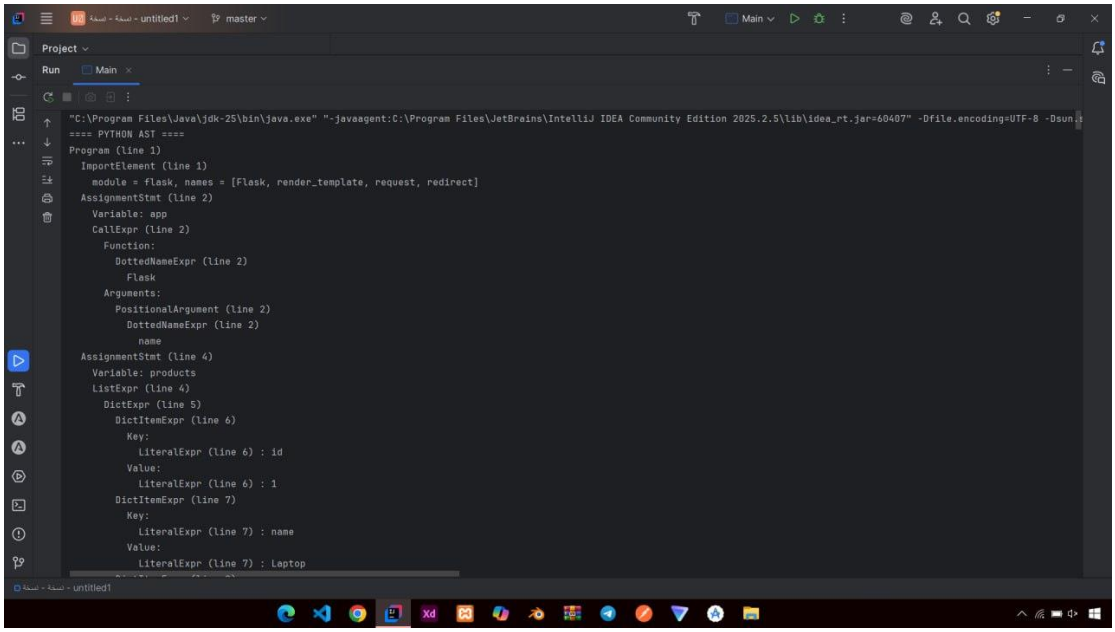
يضمن هذا الزائر تنظيم الرموز حسب Scopes بشكل صحيح، ودعم كل أنواع العبارات في البرنامج.

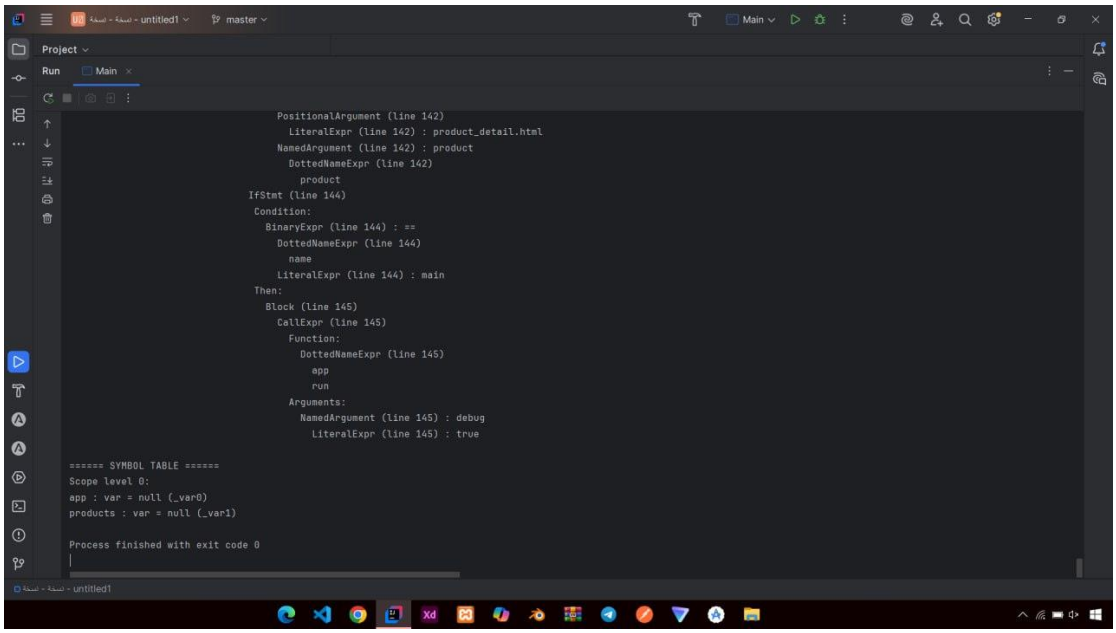
كيفية عمل Symbol Table في المشروع:

1. عند بداية البرنامج: يتم إنشاء الـ global scope.
2. عند تعريف متغير أو دالة جديدة: يتم إنشاء VariableSymbol وإضافته للـ current scope.
3. عند دخول دالة أو Block: يتم تخصيص scope جديد وربطه بالـ parent scope.
4. عند الخروج من Scope: يتم تحريره وإعادة الـ current scope إلى النطاق السابق
5. عند استخدام متغير: يتم البحث في الـ current scope ثم صعوديًا في الـ parent scopes إذا لم يُوجد محليًا.
6. تخزين المتغيرات الخاصة بالحلقات: يتم تمييزها كـ loop_var لتجنب التضارب مع متغيرات أخرى.

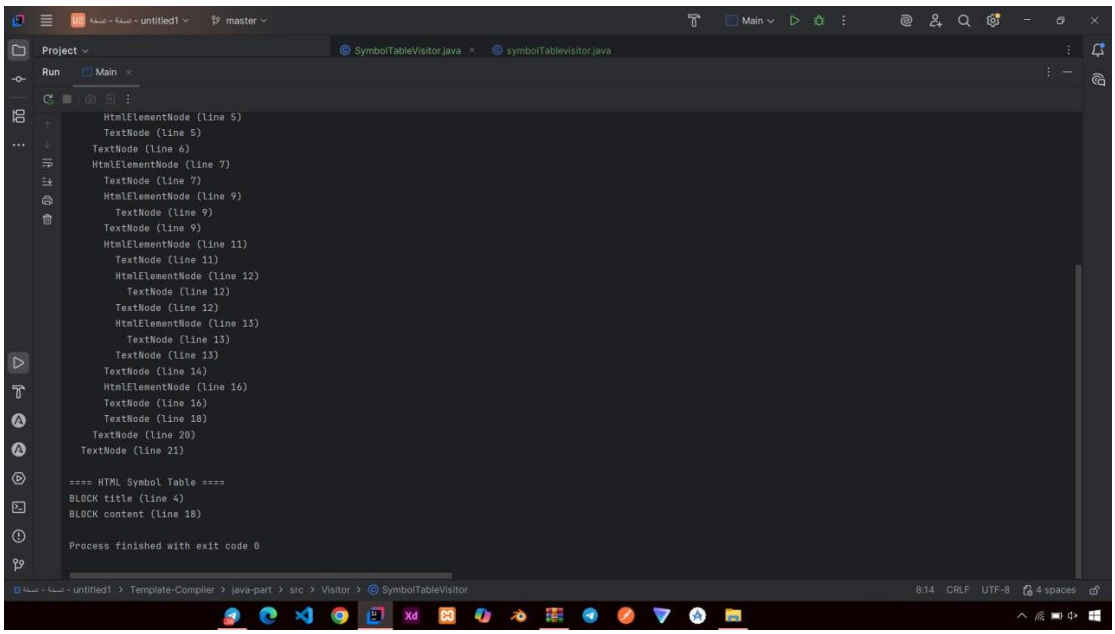
صور من الخرج :

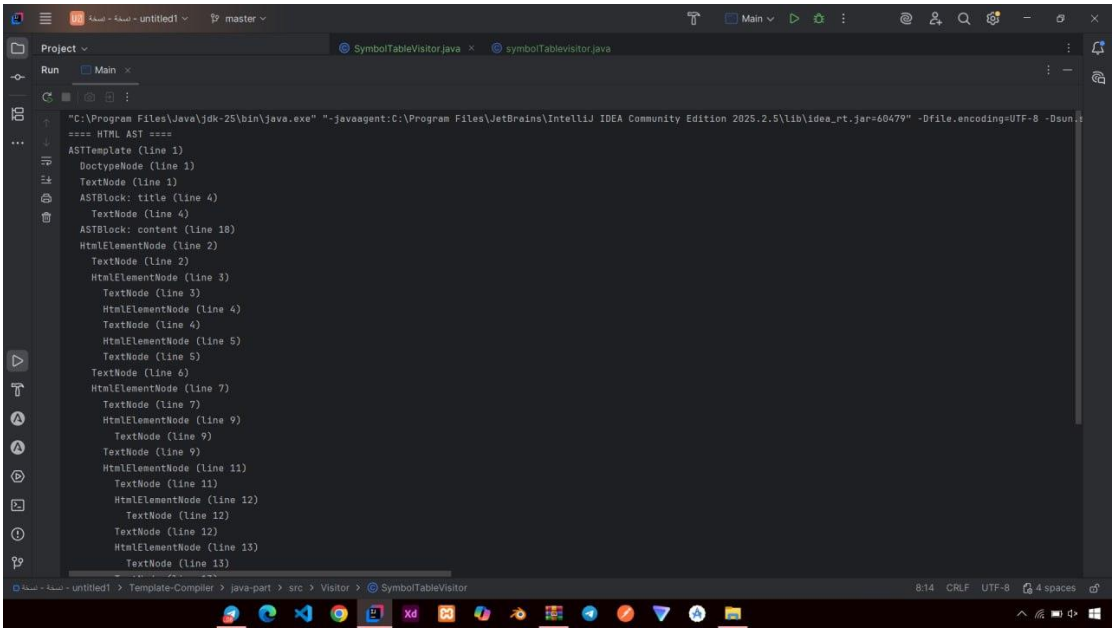
قسم البايثون :





قسم الواجهات :




"C:\Program Files\Java\jdk-25\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2025.2.5\lib\idea_rt.jar=60479" -Dfile.encoding=UTF-8 -Dsun.
==== HTML AST ====
ASTTemplate (line 1)
DoctypeNode (line 1)
TextNode (line 1)
ASTBlock: title (line 4)
TextNode (line 4)
ASTBlock: content (line 18)
HtmlElementNode (line 2)
TextNode (line 2)
HtmlElementNode (line 3)
TextNode (line 3)
HtmlElementNode (line 4)
TextNode (line 4)
HtmlElementNode (line 5)
TextNode (line 5)
TextNode (line 6)
HtmlElementNode (line 7)
TextNode (line 7)
HtmlElementNode (line 9)
TextNode (line 9)
TextNode (line 9)
HtmlElementNode (line 11)
TextNode (line 11)
HtmlElementNode (line 12)
TextNode (line 12)
TextNode (line 12)
HtmlElementNode (line 13)
TextNode (line 13)