



الجمهورية العربية السورية
جامعة دمشق

كلية الهندسة المعلوماتية

مشروع المترجمات:

أعضاء الفريق:

محمد معتز كمال القحف

يمنى جمال عثمان

علا محسن فزع

لؤي عدوان خنيفس

راما سهيل المرجي

بإشراف:

المهندس وسيم البزرة

المقدمة:

يهدف هذا المشروع إلى تصميم وتنفيذ كومبایلر يعتمد على أداة **ANTLR** لتحليل ومعالجة لغات **HTML** و **CSS** و **Jinja2**، وذلك ضمن بيئه تطوير تعتمد على لغة **Python** ومكتبة **Flask**. يركز المشروع على بناء المراحل الأساسية للكومبایلر، والتي تشمل التحليل المعجمي(**Lexer**) ، والتحليل النحوي(**Parser**) ، وبناء الشجرة المجردة(**AST**) ، واستخدام نمط التصميم **Visitor** ، بالإضافة إلى إدارة جدول الرموز(**Symbol Table**).

يساعد هذا المشروع على فهم كيفية تحليل لغات الويب من الناحية التركيبية والدلالية، كما يوضح دور أدوات توليد المحللات مثل **ANTLR** في تسهيل عملية بناء الكومبایلرات. تم اختيار لغات **HTML** و **CSS** و **Jinja2** لكونها مستخدمة بشكل واسع في تطوير تطبيقات الويب، خاصة عند دمجها مع إطار العمل **Flask** في بايثون.

يُعد هذا المشروع تطبيقاً عملياً لمفاهيم تصميم الكومبایلر، حيث يربط بين الجانب النظري والتطبيقي، ويوفر أساساً يمكن تطويره مستقبلاً لدعم التحقق من الأخطاء، أو تحسين القوالب، أو توليد مخرجات أكثر كفاءة لتطبيقات الويب.

(HTML,CSS,JINJA2)

نظرة عامة:

يُعد هذا الجزء من المشروع مسؤولاً عن التحليل النحوبي (**Syntax**)، حيث يستقبل الرموز (**Tokens**) الناتجة من الـ **Lexer** ويقوم بتنظيمها وفق قواعد نحوية محددة لبناء بنية منطقية تمثل مستندات **HTML**، بالإضافة إلى دعم عناصر **CSS** و **Jinja2**. يتم تنفيذ هذا الـ **Parser** باستخدام أداة **ANTLR** مع الاعتماد على الرموز المعرفة مسبقاً في **HTMLLexer**.

```

1  parser grammar HTMLParser;
2  @header{
3      package antlrHTML;
4  }
5  options {
6      tokenVocab=HTMLLexer;
7  }
8  root
9      :(html_content+)? EOF
10     ;
11
12 html_content
13     :tag
14     |style
15     |jinja2
16     ;
17
18 tag
19     : TAG_OPEN SLASH? tag_content SLASH? TAG_CLOSE ID*?COLON?
20     ;

```

يتم تعريف هذا الملف ك **Parser Grammar** مستقل، مع ربطه مباشرة بالـ **Lexer** عبر خيار **tokenVocab**. يتيح ذلك لـ **Parser** استخدام جميع الرموز المعرفة في **HTMLLexer** دون الحاجة لإعادة تعريفها.

تمثل هذه القاعدة **نقطة البداية (Start Rule)** للتحليل النحوی.

وتوضح أن:

- الملف يمكن أن يحتوي على واحد أو أكثر من عناصر المحتوى
(html_content)

- يمكن أن يكون الملف فارغاً

. يجب أن ينتهي التحليل بعلامة نهاية الملف EOF
هذا التصميم يمنح مرونة في تحليل ملفات HTML كاملة أو أجزاء منها.

: (html_content)--

تُعرف هذه القاعدة الأنواع المختلفة للمحتوى الذي يمكن أن يظهر داخل المستند، حيث تسمح بوجود:

. عناصر (tag).

. أنماط (style).

. أو تعليمات قوالب (Jinja2)

يساعد هذا الفصل في تنظيم التحليل النحوي والتعامل مع كل نوع من المحتوى بشكل مستقل.

: (Tag)--

تمثل هذه القاعدة البنية الأساسية لوسوم HTML ، حيث تدعم:

. الوسوم الافتتاحية والختامية

. الوسوم ذاتية الإغلاق

. مرونة في التعامل مع أسماء الوسوم والخصائص

```

22 tag_content
23   :ID ident*
24   ;
25
26 ident
27   :ID
28   |EQ
29   |DOUBLE_QUOTE_STRING
30   ;
31 //////////////style////////
32 style
33   :rule
34   | atRule
35   ;
36
37 rule : selector ID? COMMA?ID?ID?COMMA?ID?ID? LBRACE declaration* RBRACE ;
38
39 selector : simpleSelector ( combinator simpleSelector )* ( COMMA selector )? ;
40
41 simpleSelector : ( typeSelector | universal ) ( HASH | class | attrib )* ;
42
43 typeSelector : ID?;
44
45 universal : STAR ;
46
47 class : DOT ID ;
48
49 attrib
50   : LBRACK ID ( ( EQ | TILDE_EQUALS | PIPE_EQUALS | CARET_EQUALS | DOLLAR_EQUALS | STAR_EQUALS ) ( ID ) ) RBRACK #attribWithValue
51   | LBRACK ID RBRACK #attribWithoutValue
52   ;

```

:(**tag_content**)--

تمثل هذه القاعدة محتوى الوسم **HTML** ، حيث تتكون من:

- . **ID** يمثل اسم الوسم مثل (**div, p, span**) .
- . **ident*** تمثل مجموعة اختيارية من المعرفات الإضافية، مثل (**attributes**)

يسمح هذا التصميم بتحليل الوسوم البسيطة والموسعة التي تحتوي على عدد غير محدد من **الخصائص**.

:(**Ident**)--

- تُستخدم هذه القاعدة لتحليل خصائص الوسوم، حيث تدعم:
- . **أسماء الخصائص (ID)** .

. علامة الإسناد (=)

. القيم النصية داخل علامات اقتباس مزدوجة

يساعد هذا التبسيط في التعامل مع الخصائص دون تعقيد القواعد النحوية.

:**(style)**--

تمثل هذه القاعدة نقطة الدخول لتحليل أنماط CSS ، حيث تسمح إما بقواعد عادية (rule) أو قواعد خاصة (atRule) مثل @media أو

.@import

:**(rule)**--

تمثل هذه القاعدة بنية قواعد CSS الأساسية، والتي تتكون من:

. محددات (Selectors)

. كتلة تعریفات محاطة بأقواس { }

. مجموعة من التصريحات (declaration) داخل القاعدة

تم تصميم القاعدة بمرونة لدعم تنوع صيغ كتابة CSS.

:**(selector)**--

تدعم هذه القاعدة:

. المحددات البسيطة

• المحددات المركبة باستخدام combinator.

• الفصل بين محددات متعددة باستخدام الفاصلة (,) .

يسمح ذلك بتحليل سلاسل محددات CSS المعقدة.

: (simpleSelector) --

يتكون المحدد البسيط من:

• نوع العنصر (typeSelector) مثل div

• أو المحدد العام (*)

• مع دعم:

◦ المعرفات #id

◦ الأصناف class و محددات الخصائص [attr=value]

HTML يمثل اسم عنصر typeSelector ○

universal يمثل المحدد العام الذي يطابق جميع العناصر ○

• يمثل هذا المحدد الأصناف في CSS : (Class) --

: (attrib) --

تدعم هذه القاعدة نوعين من محددات الخصائص:

1. خصائص بقيمة محددة

2. خصائص بدون قيمة

تم استخدام `#attribWithValue`, `#attribWithoutValue`)
لتسهيل بناء الـ AST لاحقاً.

```

54     combinator : TAG_CLOSE #childCombinator
55         | PLUS #adjacentSiblingCombinator
56         | TILDE #generalSiblingCombinator ;
57
58     declaration : property COLON value SEMI? ;
59
60     property : ID #identProperty
61         | VAR #varProperty
62         ;
63
64     value : valuePart+ ;
65
66     valuePart : ID #identValue
67         | VAR #varValue
68         | NUMBER #numberValue
69         | HASH #hashValue
70         | IMPORTANT #importantValue
71         | URL ( ID | NUMBER | SLASH | COLON | DOT | QUESTION | EQ | AND | MINUS | PLUS )* RPAREN #urlValue
72         | LPAREN value RPAREN #parenValue
73         | COMMA #commaValue
74         | DOT #dotValue
75         | COLON #colonValue
76         | SLASH #slashValue
77         | TAG_OPEN #greaterValue
78         | PLUS #plusValue
79         | MINUS #minusValue
80         | STAR #starValue
81         | TILDE #tildeValue
82         | EQ #equalsValue
83         | QUESTION #questionValue
84         | DOUBLE_QUOTE_STRING #doubleQuoteValue
85         ;

```

:combinator--

تُستخدم هذه القاعدة لتعريف العلاقات بين محددات CSS ، حيث تدعم:

childCombinator (>) .

adjacentSiblingCombinator (+) .

generalSiblingCombinator (~) .

عام

تم استخدام Labels لتسهيل التمييز بين أنواع العلاقات أثناء بناء AST.

: (Declaration) --

تمثل هذه القاعدة تعريف خاصية CSS واحدة، حيث:

• تتكون من اسم الخاصية (property)

• متبوعة بعلامة النقطتين (:)

• ثم القيمة (value)

• مع دعم الفاصلة المنقوطة الاختيارية (;)

هذا التصميم يتماشى مع مرونة صياغة CSS.

: (Property) --

تدعم هذه القاعدة نوعين من الخصائص:

• خصائص قياسية مثل color, margin

• خصائص معرفة باستخدام متغيرات CSS

استخدام Labels يسهل التمييز الدلالي أثناء المعالجة اللاحقة.

: (Value) --

تمثل هذه القاعدة قيمة خاصية CSS ، والتي قد تتكون من:

• جزء واحد أو عدة أجزاء

• مما يسمح بدعم قيم معقدة.

: (ValuePart) --

تُعد هذه القاعدة من أكثر القواعد شمولًا، حيث تدعم:

- القيم النصية
- الأرقام مع الوحدات
- الألوان
- الروابط
- العمليات الحسابية
- القيم المحاطة بأقواس
- النصوص بين علامات اقتباس

كل نوع من القيم يحمل **Label** خاص به، مما يجعل بناء الـ AST و معالجة القيم لاحقًا أكثر تنظيمًا و دقة.

```

87  atRule
88      : AT_RULE ( valuePart | URL ( ID | NUMBER )? RPAREN )* atRuleBody
89      ;
90
91  atRuleBody
92      : LBRACE ( rule | declaration | atRule )* RBRACE #atRuleWithBlock
93      | SEMI #atRuleWithoutBlock
94      ;

```

: (Property) --

تمثل هذه القاعدة قواعد CSS الخاصة (At-Rules) مثل:

@media .

@import .

@font-face .

@keyframes .

شرح مكونات القاعدة:

@AT_RULE يمثل الكلمة المحفوظة التي تبدأ بـ @ .

. الجزء الأوسط يسمح بوجود:

- قيم عامة (valuePart)

- أو دوال مثل (url(...))

atRuleBody يمثل جسم القاعدة .

هذا التصميم يسمح بدعم الصيغ المختلفة لقواعد At-Rule في CSS.

:**(atRuleBody)--**

تدعم هذه القاعدة نوعين من At-Rules:

: (Block) مع كتلة At-Rule ◆

يتم تمثيل هذا النوع باستخدام:

. أقواس { } .

. محتوى داخلي قد يشمل:

◦ قواعد CSS .

◦ تصريحات .

◦ أو At-Rules أخرى (تعشيش)

: At-Rule بدون كتلة ◆

يتم تمثيله باستخدام فاصلة منقطة فقط.

استخدام Labels (#atRuleWithBlock, #atRuleWithoutBlock)

يسهل التمييز بين النوعين أثناء بناء الشجرة المجردة (AST).

```

97     jinja2
98         :statement
99
100        ;
101    statement
102        : stmt|
103        | expr
104        ;
105    stmt
106        :BLOCK_START ID*? BLOCK_END
107        ;
108
109    expr
110        :VAR_START expr_content*  VAR_END
111        ;
112    expr_content
113        :ID
114        |LPAREN
115        |RPAREN
116        |COLON
117        |NUMBER
118        |DOUBLE_QUOTE_STRING
119        |OR
120        |LBRACK
121        |RBRACK
122        |PIPE
123        |TILDE
124        |TAG_CLOSE
125        ;

```

--(Jinja2): تمثل هذه القاعدة نقطة الدخول لتحليل أي تعليمة Jinja2 داخل المستند، حيث يتم تمرير المعالجة إلى قاعدة .statement

:(statement)--

تُقسم تعليمات Jinja2 إلى نوعين رئيسيين:

تعليمات تحكم أو أوامر **Statements:** .

عبارات يتم تقييمها وإخراج نتائجها **Expressions:** .

هذا التقسيم يعكس البنية الأساسية للغة `Jinja2`.

Block : تمثل هذه القاعدة تعليمات `Jinja2` من نوع `Block` **(stmt)--Statement**

`%BLOCK_START` يمثل : %

`ID*`? يسمح بوجود كلمات أو معرفات داخل التعليمة .

`%BLOCK_END` يمثل : %

تم تبسيط القاعدة لدعم أشكال متعددة من التعليمات دون تعقيد نحوبي.

:(expr)--

تُستخدم هذه القاعدة لتحليل العبارات التي تُكتب داخل:

`VAR_START` يمثل : {

`VAR_END` يمثل : }

`expr_content*` يسمح بسلسلة مرنة من الرموز داخل التعبير .

:**(expr_content)--**

تدعم هذه القاعدة العناصر الأساسية التي يمكن أن تظهر داخل تعبيرات **Jinja2**، مثل:

- المعرفات والمتغيرات
- الأرقام
- النصوص
- العمليات المنطقية
- الفلاتر (|)
- الأقواس والمحددات

هذا يسمح بتحليل تعبيرات ديناميكية مستخدمة بكثرة في قوالب **Flask**.

Jinja2 ، HTML ، CSS تحليل

مقدمة

يهدف هذا المشروع إلى تصميم وتنفيذ كومبایلر باستخدام **ANTLR** لتحليل ملفات الويب التي تحتوي على **HTML**، **CSS**، و**قوالب Jinja2**. يعتمد المشروع على **Python** ويستهدف تحسين فهم البنية التركيبية والدلالية للصفحات الديناميكية، مع إمكانية التوسيع لدعم التطبيقات المبنية باستخدام **Flask**. العمل إطار.

HTML تحليل 1

تم تصميم Parser وLexer لدعم بنية HTML الأساسية:

- يقوم بتحويل النص إلى رموز أساسية مثل , <, >, {, }, . معرفات الوسوم، النصوص، التعليقات، والمتغيرات.

يحدد البنية التركيبية للصفحات من خلال قواعد مثل:

نقطة البداية لملف root :

يحدد أن المحتوى يمكن أن يكون وسم html_content :

Jinja2 ، أو CSS ، أو تعليمات HTML

تمثل الوسوم وخصائصها tag_content :

يساعد هذا التحليل في بناء شجرة بنية (AST) للصفحات، مما يسهل المراحل اللاحقة من التحقق الدلالي ومعالجة المتغيرات.

CSS تحليل 2

يدعم المشروع قواعد CSS التقليدية والمتقدمة:

Rule Parsing:

تحدد عناصر CSS والحدادات المرتبطة selector : rule . بها.

تحليل الخصائص والقيم value و property declaration . المختلفة.

At-Rules: .

@media, @ruleBody : gatRule .
مع أو بدون كتل.

Combinators: .

تدعم علاقات مثل ~, +, < بين العناصر.

Flexibility: .

يدعم الألوان، المتغيرات، الروابط، النصوص، العمليات الحسابية، والقيم المعقدة.

استخدام Labels في القواعد يسهل لاحقاً بناء AST وتصنيف أنواع القيم والخصائص.

3 تحليل Jinja2

يتيح المشروع تحليل قوالب Jinja2 المدمجة داخل HTML:

Statements: .

{% if %}, {% for %}. stmt : تعليمات التحكم مثل .

Expressions: .

{{ variable }}. expr : تعبيرات ديناميكية مثل .

expr_content: .

- دعم المتغيرات، الأرقام، النصوص، الأقواس، العمليات المنطقية والفلاتر.

يسمح هذا بتحليل الصفحات الديناميكية، مما يجعل الكومبايلر قادرًا على التعامل مع المحتوى المولد برمجيًا في تطبيقات Flask.

الخلاصة

تمثل هذه المرحلة من المشروع **الجزء الأمامي للكومبايلر**، حيث تم بناء قادرین على التعامل مع **HTML**، **CSS**، و **Jinja2** بشكل Parser وLexer متكامل.

يتيح هذا التصميم:

- استخراج **Tokens** دقيقة من النص
- تحليل بنية المستندات لإنشاء **AST**
- دعم التعبيرات الديناميكية في صفحات الويب
- توفير أساس قوي للمرحلة التالية من المشروع، والتي ستشمل دعم **Python** و **Flask**

(Python,Flask)

```

1  parser grammar PythonParser;
2  > @header{...}
3  options { tokenVocab = PythonLexer; }
4
5  root
6      : ((NEWLINE | stmt)+)? EOF
7      ;
8
9
10 stmt
11     : simple_stmt
12     | compound_stmt
13     | decorated
14     ;
15
16 simple_stmt
17     : small_stmt (SEMI small_stmt)* SEMI? NEWLINE          #simple_stmt_line
18     | LBRACE small_stmt (COMMA small_stmt)*? RBRACE (COMMA + NEWLINE + simple_stmt)*? #simple_stmt_block
19     ;
20
21 decorated
22     : decorator+ compound_stmt
23     ;
24
25 compound_stmt
26     : (IF | ELIF | ELSE) test COLON suite                  # if_stmt
27     | WHILE test COLON suite                            # while_stmt
28     | FOR exprlist IN (COMMA? test)* COLON suite       # for_stmt
29     | TRY COLON suite (except_clause+)                 # try_stmt
30     | WITH with_item (COMMA with_item)* COLON suite   # with_stmt
31     | DEF name LPAREN (name COMMA?)* RPAREN (ARROW test)? COLON suite # func_def
32     ;
33
34

```

: (root)--

. تمثل نقطة البداية لتحليل الملف.

. تقبل أي عدد من الأسطر الفارغة (NEWLINE) أو التعليمات . قبل نهاية الملف (stmt)

. تسمح بمرونة في التعامل مع الملفات الفارغة أو الملفات التي تحتوي على عدة جمل متتالية.

:stmt--

. تحدد أن كل جملة في Python يمكن أن تكون:

◦ جملة بسيطة (simple_stmt).

◦ جملة مركبة (compound_stmt).

◦ جملة مزخرفة (decorated).

. هذا التصميم يعكس الطبيعة الهيكلية للغة Python ، حيث أن كل كود يمكن تصنيفه ضمن أحد هذه الأنواع.

:simple_stmt--

. النوع الأول: (#simple_stmt_line):

◦ سلسلة من التعليمات الصغيرة (small_stmt) مفصولة بفواصل
منقوطة ;

◦ تنتهي بالسطر الجديد (NEWLINE).

. النوع الثاني: (#simple_stmt_block):

◦ يسمح بكتل تعليمية محاطة بأقواس { }

◦ تدعم فصل التعليمات بفواصل، مع السماح بالاستمرار في
الأسطر التالية.

. الهدف: التعامل مع الجمل القصيرة أو المجمعة بطريقة مرنة.

: (decorated) --

- . تمثل الدوال أو الكائنات المزخرفة بـ Python.
- . يسمح بتطبيق أكثر من زخرفة واحدة decorator+ .
- . يتم تطبيق الزخرفة على جملة مركبة (compound_stmt).

: (compound_stmt) --

- . if, elif, else: تحليل جمل الشرطية.
- . while: تحليل حلقات التكرار.
- . for_stmt : تحليل حلقات التكرار مع دعم التعبيرات المتعددة في القيم.
- . try_stmt : تحليل كتلة try مع أكثر من except_clause.
- . with_stmt : تحليل جمل with مع دعم أكثر من عنصر.
- . func_def : تحليل تعريف الدوال def، مع دعم الزخرفة الاختيارية، القوسين، القيم الافتراضية، وتحديد نوع الإرجاع ->
- . كل جملة مركبة تحمل Label خاص لتسهيل بناء الـ AST وتحديد نوع الجملة أثناء التحليل الدلالي.

```

35     suite
36         : simple_stmt
37         | INDENT stmt+ DEDENT
38         | NEWLINE stmt*
39         ;
40
41     decorator
42         : AT (DOT? name)* (LPAREN (COMMA? test)* RPAREN)? NEWLINE
43         ;
44
45     with_item
46         : test (AS expr)?
47         ;
48
49     except_clause
50         : EXCEPT COLON suite
51         ;
52
53     exprlist
54         : expr (COMMA expr)* COMMA?
55         ;
56
57     small_stmt
58         : test assign_part?                                # expr_stmt
59         | RETURN (COMMA?test)*                            # return_stmt
60         | IMPORT name                                    # import_stmt
61         | FROM name IMPORT ( COMMA? name)*             # from_stmt
62         ;
63
64
65
66
67

```

:(suite)--

تمثل هذه القاعدة مفهوم الكتلة (Block) في لغة Python ، حيث تدعم:

- . جملة واحدة بسيطة
- . كتلة متعددة الجمل تعتمد على المسافات الباردة ()
- . (DEDENT/

. كتلة تبدأ بسطر جديد يتبعها مجموعة من الجمل

تعكس هذه القاعدة اعتماد Python على Indentation بدل الأقواس لتحديد البلوكات.

: (decorator)--

تُستخدم هذه القاعدة لتحليل **Decorators** في Python ، والتي تُطبق على الدوال أو الأصناف.

- . @بداية الزخرفة

- . دعم الأسماء المتسلسلة مثل @module.decorator

- . دعم استدعاء الزخرفة مع معاملات

- . تنتهي بسطر جديد

: (with_item)--

تمثل هذه القاعدة عنصراً واحداً داخل جملة **with**، حيث:

- . يتم تحليل التعبير الأساسي

- . مع دعم تعيين النتيجة إلى متغير باستخدام **as**

: (except_clause)--

تحل هذه القاعدة جمل معالجة الأخطاء **except**، حيث:

- . تبدأ بالكلمة الممحوزة **except**

- . يتبعها جسم الكتلة البرمجية الخاصة بمعالجة الاستثناء

: (exprlist)--

تمثل هذه القاعدة قائمة من التعبيرات، وتُستخدم في:

- . حلقات **for**.
 - . عمليات التعيين المتعددة.
 - . تمرير عدة معاملات.
- تدعم الفاصلة الختامية الاختيارية.

: (small_stmt)--

تحدد هذه القاعدة الجمل الأساسية في Python:

- expr_stmt**: تعبير أو تعيين قيمة .
 - return_stmt**: إرجاع قيمة من دالة .
 - import_stmt** : استيراد مكتبة .
 - from_stmt** : استيراد عناصر محددة من مكتبة .
- كل نوع يحمل **Label** خاص لتسهيل التحليل الدلالي وبناء الـ AST.

```

64 assign_part
65   :COLON test (EQ (COMMA? test)*)?                                #annotatedAssign
66   | EQ LBRACK NEWLINE simple_stmt NEWLINE RBRACK                  #listAssign
67   | EQ LBRACE NEWLINE small_stmt (COMMA small_stmt)*NEWLINE RBRACE  #blockAssign
68   ;
69
70
71 test
72   : comparison
73   | NOT test
74   | test AND test
75   | test OR test
76   ;
77
78 comparison
79   : expr ( ( EQUALS | NOT_EQ_2 | (NOT? IN) | (IS (NOT)?) | EQ ) expr )*
80   ;
81
82 expr
83   : atom trailer*          #atomExpr
84   | expr (PLUS | MINUS) expr #additiveExpr
85   ;
86
87 atom
88   : LPAREN testlist_comp? RPAREN      #parenAtom
89   | LBRACK testlist_comp? RBRACK    #listAtom
90   | name                         #nameAtom
91   | PRINT                         #printAtom
92   | MINUS? number                 #numberAtom
93   | NONE                          #noneAtom
94   | STRING+                      #stringAtom
95   .

```

:**(assign_part)--**

تُستخدم هذه القاعدة لتحليل أنواع مختلفة من عمليات التعيين في **Python**

annotatedAssign .

يدعم التعيين مع توصيف نوع المتغير (Type Annotation)

listAssign .

يسمح بتعيين قيمة داخل قائمة متعددة الأسطر.

blockAssign .

يدعم تعيين كتل متعددة التعلميات داخل أقواس {} . يساعد هذا التنوع في دعم صيغ تعيين متقدمة.

:test)--

تمثل هذه القاعدة **العبارات المنطقية**، حيث تدعم:

- . المقارنات
- . العمليات المنطقية not, and, or
- . التعبيرات المركبة والمتداخلة

:(comparison)--

تدعم هذه القاعدة عمليات المقارنة في Python ، مثل:

- = , != .
 - in, not in .
 - is, is not .
- . المقارنات المتسلسلة

: (expr)--

تحلل هذه القاعدة التعبيرات الحسابية:

- تعبيرات بسيطة تعتمد على الذرات (atom)
- عمليات الجمع والطرح
- دعم استدعاءات متسلسلة عبر trailer

: (atom)--

تمثل الذرات الوحدات الأساسية للتعبير في Python ، وتشمل:

- تعبيرات بين أقواس
- القوائم
- المتغيرات
- الأرقام
- القيم الفارغة (None)
- النصوص

كل نوع معرف بـ **Label** لتسهيل بناء الـ **AST**

```

97  testlist_comp
98      : test (FOR test IN (COMMA? test)* (IF test)* | (COMMA test )*)
99      ;
100
101 name
102     : NAME
103     | TRUE
104     | FALSE
105     ;
106
107 number
108     : DECIMAL_INTEGER
109     | FLOAT_NUMBER
110     ;
111
112 trailer
113     : DOT name
114     | LPAREN (COMMA? test)* RPAREN
115     ;
116

```

:**(testlist_comp)--**

تمثل هذه القاعدة قوائم القيم (Lists) و **List Comprehensions** في

:Python

- . تدعم القوائم التقليدية

- . وتدعم تعبيرات الفهم: **(Comprehension)**

يسمح هذا التصميم بتحليل الصيغ البسيطة والمعقدة بنفس القاعدة.

: (name) --

تحل هذه القاعدة:

• أسماء المتغيرات

• القيم المنطقية (True, False)

وبذلك تميز بين المعرفات والقيم الثابتة.

: (number) --

تدعم هذه القاعدة:

• الأعداد الصحيحة

• الأعداد العشرية

وستُستخدم في جميع السياقات الحسابية والمنطقية.

: (trailer) --

تُستخدم هذه القاعدة لتحليل الوصول إلى الخصائص واستدعاء الدوال:

• :DOT name

• للوصول إلى خصائص الكائنات

• :LPAREN ... RPAREN

• لاستدعاء الدوال مع معاملات

تحليل لغة Python ودمجها مع :Flask

في هذا الجزء من المشروع تم تطوير **Parser** للغة **Python** باستخدام أداة **ANTLR**، اعتماداً على **Lexer** مخصص يدعم البنية الحقيقية للغة، بما في ذلك الكلمات المحوزة، المعاملات، النصوص، والأرقام، بالإضافة إلى التعامل مع **Indentation** من خلال رموز **DEDENT** أو **INDENT** لمحاكاة طريقة **Python** في تحديد الكتل البرمجية.

تحليل لغة Python

تم تصميم **Parser** لتحليل البنية النحوية للغة **Python** بشكل منظم، حيث يدعم:

- الجمل البسيطة مثل:

 - التعبير
 - أوامر **return**
 - أوامر الاستيراد **from** و **import**

- الجمل المركبة مثل:

 - الشروط **if** / **elif** / **else**
 - الحلقات **while** و **for**
 - تعریف الدوال **def**
 - جمل **try** / **except** و **with**

- الكتل البرمجية المعتمدة على المسافات الباردة(**Indentation**)

- . الزخارف (Decorators)
 - . عمليات التعيين البسيطة والمتقدمة
 - . التعبير الحسابية والمنطقية
 - . القوائم وتعبيرات الفهم (List Comprehension)
 - . استدعاء الدوال والوصول إلى خصائص الكائنات
- يتم تحويل الكود بعد التحليل النحوى إلى تمثيل شجري (AST) يُستخدم لاحقًا في مراحل التحليل الدلالي والتنفيذ.
-

دمج Flask

بعد تحليل كود Python ، تم دعم استخدام مكتبة **Flask** لبناء تطبيقات ويب ديناميكية، حيث يسمح المشروع بتحليل ملفات Python التي تحتوي على:

- . تعریف المسارات (Routes)
- . استخدام الزخارف مثل @app.route
- . ربط الدوال بالواجهات
- . تمرير البيانات بين الخادم والقوالب

يساهم هذا الدمج في إنشاء بيئة متكاملة تجمع بين تحليل لغة Python وتنفيذها في سياق تطبيقات ويب.

أهمية هذا الجزء

يُوفّر هذا الجزء من المشروع:

- دعماً حقيقياً لبنيّة لغة Python.

- إمكانية تحليل تطبيقات Flask بشكل منظم.

- أساساً قوياً لربط منطق الخادم مع قوالب HTML و Jinja2.

مما يجعل النظام قادرًا على التعامل مع تطبيقات ويب كاملة بدءًا من التحليل اللغوي وحتى التنفيذ.

الشجرة المجردة (AST)

مقدمة

بعد إتمام مرحلة التحليل النحوی (Parsing) باستخدام ANTLR ، تأتي مرحلة بناء الشجرة المجردة (**Abstract Syntax Tree – AST**) ، والتي تهدف إلى تمثيل بنية البرنامج بشكل منطقي ومجرد، بعيداً عن التفاصيل النحوية مثل الأقواس، الفوائل، والرموز الزائدة.

في هذا المشروع تم تصميم وتنفيذ **AST** مستقل لكل من:

- . لغة Python , Flask .

- . Jinja2 و CSS و HTML .

بما يتيح التعامل مع كل لغة حسب طبيعتها ودورها داخل تطبيقات الويب.

تصميم AST

تم اعتماد أسلوب **Node-based Design**، حيث تمثل كل عقدة (Node) عنصراً لغوياً محدداً، مثل:

- . تعليمات .

- . تعابير .

- . كتل .

• تعریفات

• عناصر واجهة

ويتم إنشاء هذه العقد أثناء المرور على شجرة الـ Parse Tree الناتجة من الـ Parser.

Python لغة AST

يدعم AST الخاص بلغة Python تمثيل العناصر التالية:

• الجمل البسيطة والمركبة

• تعريف الدوال والمعاملات

• العبارات الشرطية والحلقات

• عمليات التعين

• التعبير الحسابية والمنطقية

• استدعاء الدوال والوصول إلى الخصائص

يساعد هذا التمثيل في تحليل منطق البرنامج وفصل البنية البرمجية عن الصيغة النحوية.

HTML-CSS AST

تم تصميم AST خاص لتمثيل:

• عناصر HTML (Tags)

. الخصائص (Attributes)

. بنية التداخل (Nesting)

. قواعد CSS

. المحددات (Selectors)

. الخصائص والقيم

يسمح هذا التمثيل بفهم بنية الواجهة بشكل هرمي، مما يسهل تحليل الصفحات والتفاعل معها برمجياً.

Jinja2-AST

يدعم AST الخاص بـ:

. المتغيرات {{}}

. العبارات الشرطية

. الحلقات

. الكتل (Blocks)

. أوامر الوراثة والتضمين (extends, include)

ويُستخدم هذا التمثيل لربط منطق الخادم مع الواجهة الأمامية بطريقة منظمة.

Flask-AST

تم بناء AST لتمثيل عناصر Flask الأساسية، مثل:

- تعريف المسارات (Routes).

- الزخارف (Decorators).

- دوال المعالجة (View Functions).

- ربط الدوال بالقوالب.

يساعد هذا الجزء في تحليل تطبيقات الويب وفهم العلاقة بين المسارات والمنطق البرمجي.

أهمية AST في المشروع

يوفر استخدام AST في هذا المشروع:

- فصلاً واضحاً بين التحليل النحوی والتنفيذ.

- بنية موحدة للتحليل الدلالي.

- سهولة تطبيق نمط Visitor.

- إمكانية التحقق من الأخطاء.

- قابلية التوسيع لاحقاً.

الخلاصة

يمثل الـ AST المرحلة الأساسية التي تربط بين تحليل اللغات المختلفة داخل المشروع، حيث يسمح بالتعامل مع Python ، HTML ، CSS ، Jinja2 و Flask ضمن نموذج موحد، مما يجعل النظام قادرًا على تحليل وبناء تطبيقات ويب متكاملة بشكل منظم وقابل للتطوير.

نط الزائر (Visitor Pattern)

مقدمة

بعد بناء الشجرة المجردة(AST) ، تم استخدام نمط الزائر (Visitor Pattern) لتنفيذ العمليات على الشجرة دون تعديل بنيتها. يُعد هذا النمط من أكثر الأنماط استخداماً في تصميم المترجمات، حيث يسمح بفصل البيانات (AST) عن العمليات المنفذة عليها.

في هذا المشروع تم تنفيذ زائرين مستقلين لمعالجة لغات الواجهة والخادم بشكل منفصل.

1 Visitor الخاص بـ Python و Flask

يُستخدم هذا الزائر لمعالجة منطق الخادم، ويقوم بـ:

- . المرور على عقد AST الخاصة بلغة Python

- . تحليل تعريف الدوال

- . معالجة الزخارف (Decorators) الخاصة بـ Flask

• استخراج تعریفات المسارات (Routes)

• ربط المسارات بدوال المعالجة

• تحليل استدعاءات القوالب وتمرير المتغيرات

يسمح هذا الزائر بفهم البنية الوظيفية لتطبيق Flask وربطها بالواجهة الأمامية.

2 Visitor الخاص بـ HTML و CSS و Jinja2

يختص هذا الزائر بمعالجة الواجهة الأمامية، حيث يقوم بـ:

• زيارة عناصر HTML وبناء الهيكل الهرمي للصفحة

• تحليل خصائص العناصر (Attributes)

• معالجة قواعد CSS والمحددات

• تفسير متغيرات Jinja2

• تحليل العبارات الشرطية والحلقات داخل القوالب

يتيح هذا الفصل التعامل مع الواجهة الأمامية بطريقة منظمة ومستقلة عن منطق الخادم.

3 آلية العمل

يعتمد كل Visitor على:

- زيارة العقد من الأعلى إلى الأسفل (Depth-First Traversal)
- تنفيذ منطق خاص بكل نوع عقدة
- تجميع المعلومات أثناء المرور
- تمرير النتائج إلى المراحل اللاحقة (Symbol Table) أو Execution

يتم اختيار الزائر المناسب حسب نوع الملف أو السياق البرمجي.

فوائد استخدام Visitor 4

يساهم استخدام نمط الزائر في:

- فصل منطق التنفيذ عن بنية البيانات
 - سهولة إضافة عمليات جديدة دون تعديل AST
 - تحسين قابلية الصيانة
 - دعم تعدد اللغات داخل مشروع واحد
 - تنظيم واضح بين منطق الخادم والواجهة
-

الخلاصة

يمثل الـ Visitor المرحلة التشغيلية الأساسية في المشروع، حيث يتم من خلاله تحويل الـ AST إلى سلوك فعلي. وقد أتاح وجود زائرين مستقلين التعامل مع Python و Flask من جهة،

و HTML و CSS و Jinja2 من جهة أخرى، مما جعل تصميم النظام أكثر مرونة و قابلية للتوسيعة.

جدول الرموز (Scopes) وإدارة النطاقات (Symbol Table)

مقدمة

بعد بناء الشجرة المجردة (AST) وتنفيذ المروور عليها باستخدام نمط الزائر (Visitor Pattern)، تم تنفيذ جدول الرموز (Symbol Table) ليكون مسؤولاً عن تخزين وإدارة المعلومات الدلالية الخاصة بالمعرفات المستخدمة في المشروع.

يشكّل جدول الرموز عنصراً أساسياً في مرحلة التحليل الدلالي (Semantic Analysis)، حيث يضمن صحة استخدام المتغيرات، الدوال، والعناصر المختلفة ضمن نطاقاتها الصحيحة.

في هذا المشروع تم إنشاء جدولي رموز منفصلين:

Flask و Python خاص بـ **Symbol Table** .

Jinja2 و CSS و HTML خاص بـ **Symbol Table** .

لـ **1** **Symbol Table** **و Flask و Python**

يختص هذا الجدول بإدارة رموز منطق الخادم، حيث يقوم بتخزين:

- . أسماء المتغيرات
- . تعریفات الدوال
- . المعاملات
- . الزخارف (Decorators)

• مسارات (Routes) .

إدارة النطاقات (Scopes)

تم استخدام نظام Scopes متداخلة يعكس البنية الحقيقية للغة Python:

• نطاق عام (Global Scope)

• نطاق الدوال

• نطاق الكتل الشرطية والحلقات

يتم إنشاء Scope جديد عند الدخول إلى دالة أو كتلة، وإزالته عند الخروج منها، مما يضمن:

• منع تعارض الأسماء

• دعم الإخفاء (Shadowing)

• التحقق من صحة الوصول إلى المتغيرات

2 - Jinja2 و CSS و HTML و Symbol Table

يُستخدم هذا الجدول لإدارة رموز الواجهة الأمامية، ويقوم بتخزين:

• معرفات عناصر HTML

• أسماء الكلاسات

• خصائص CSS

• متغيرات Jinja2

. الكتل (Blocks) والقوالب الموروثة

إدارة النطاقات

تم اعتماد Scopes تعكس طبيعة القوالب:

. نطاق الصفحة

. نطاق الكتل (Blocks)

. نطاق الحلقات والشروطيات داخل Jinja2

يساعد هذا النظام على:

. التحقق من استخدام المتغيرات داخل القالب

. ضمان توافق الوراثة بين القوالب

. منع التعارض بين المتغيرات المحلية وال العامة

3 التكامل مع Visitor و AST

يتم ملء جداول الرموز أثناء المرور على الـ AST باستخدام الزائر المناسب، حيث:

. يتم إدخال الرموز عند تعريفها

. يتم البحث عنها عند استخدامها

. يتم التحقق من النطاق المناسب

يسمح هذا التكامل بالكشف المبكر عن الأخطاء الدلالية.

فوائد استخدام Scopes مع Symbol Table 4

يوفر هذا التصميم:

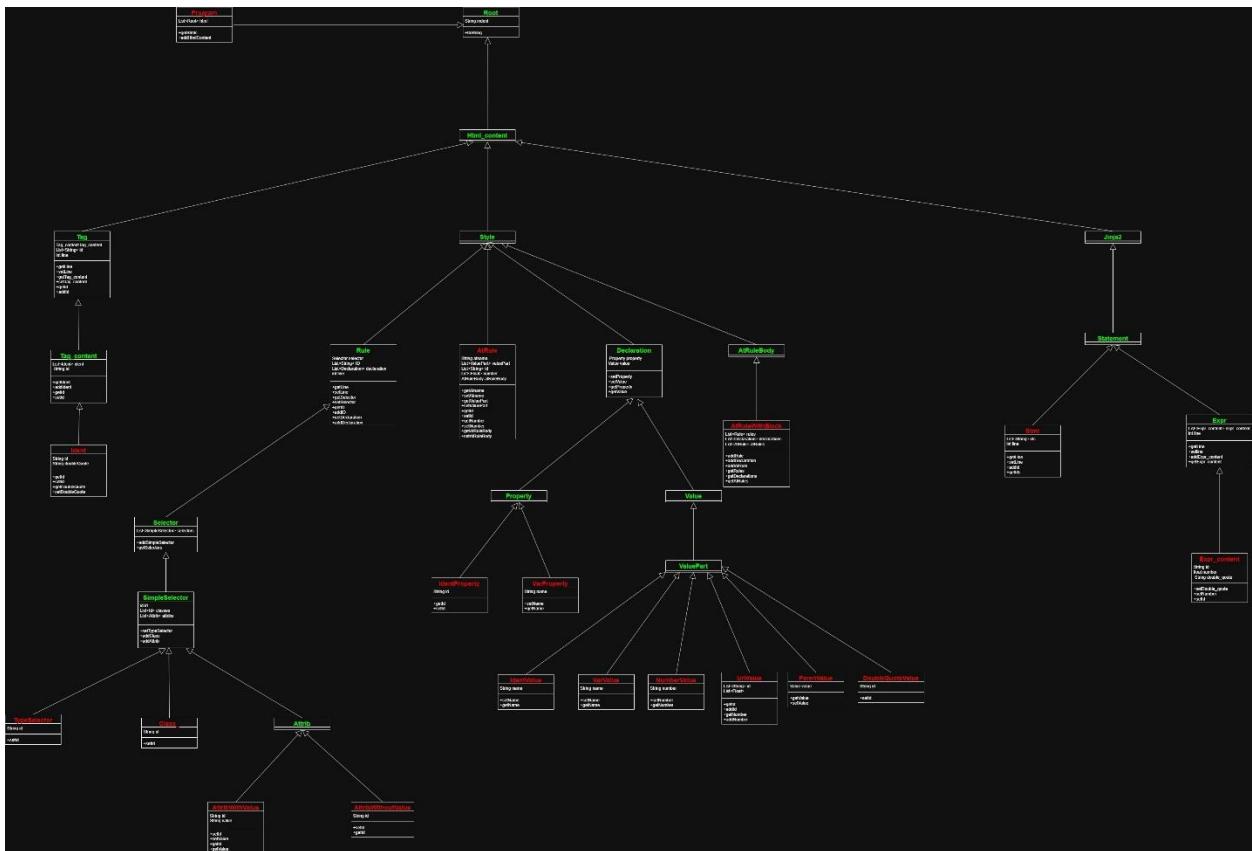
- تنظيماً دقيقاً للرموز
- دعماً لتنوع اللغات داخل المشروع
- فصلاً واضحاً بين منطق الخادم والواجهة
- سهولة التوسيع مستقبلاً
- بنية قوية للتحقق الدلالي

الخلاصة

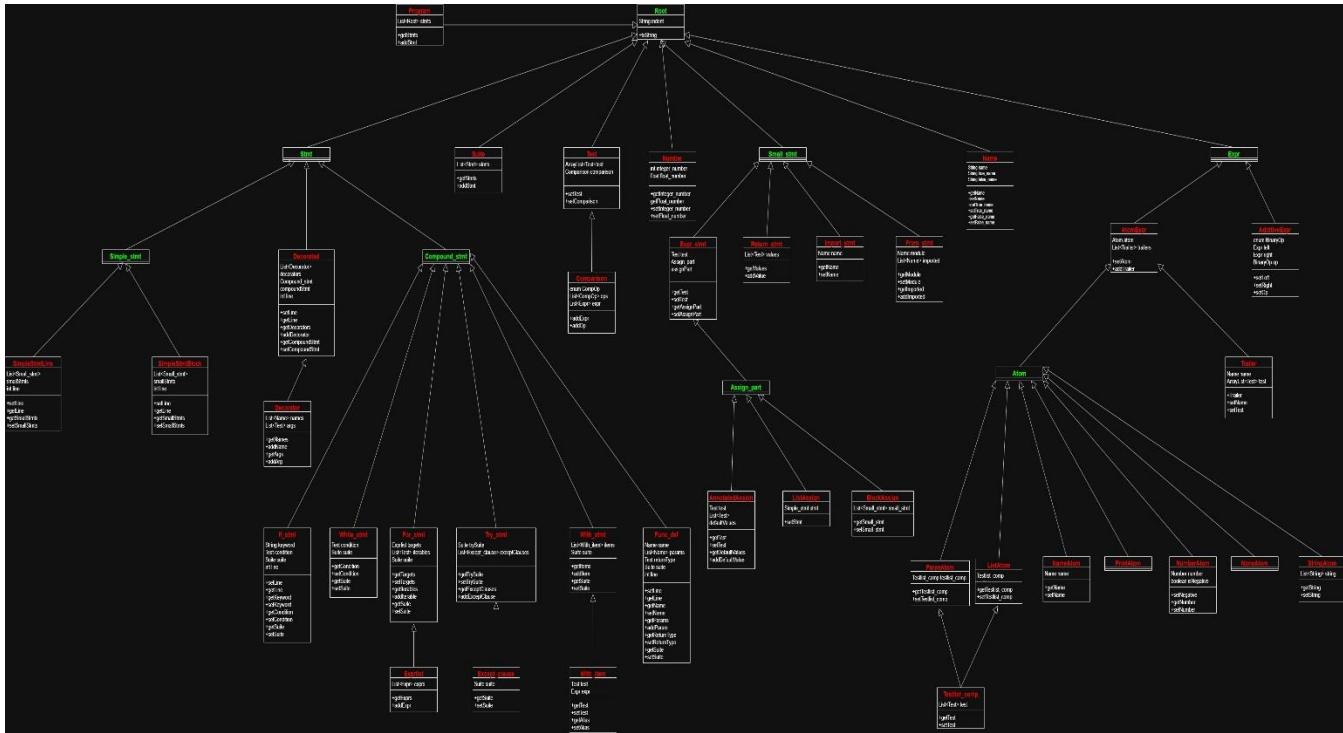
يعد جدول الرموز مع دعم النطاقات أحد أهم مكونات هذا المشروع، حيث ممكن من إدارة المعرفات بشكل صحيح لكل من Python/Flask و HTML/CSS/Jinja2.

وقد ساهم هذا الفصل في تحقيق تكامل حقيقي بين اللغات المختلفة ضمن بيئة واحدة قابلة للتحليل والتنفيذ.

HTML diagram



Python diagram



الخاتمة

في ختام هذا المشروع، تم بناء نظام متكامل يحاكي مراحل عمل المترجم (Compiler) بشكل عملي ومنهجي، حيث شمل المشروع جميع المراحل الأساسية بدءاً من التحليل المعجمي (Lexer)، مروراً بالتحليل النحوي (Parser)، وبناء الشجرة المجردة (AST)، ثم تنفيذ العمليات عليها باستخدام نمط الزائر (Visitor Pattern)، وانتهاءً بمرحلة التحليل الدلالي من خلال جداول الرموز (Symbol Table) مع دعم النطاقات (Scopes).

تميز المشروع بدعمه لعدة لغات وتقنيات مستخدمة في تطوير تطبيقات الويب، وهي **Flask** و **Python** و **HTML** و **CSS** و **Jinja2** في جانب الخادم، مما أتاح فهماً عميقاً لكيفية تكامل هذه اللغات داخل تطبيق ويب واحد. وقد ساهم هذا الفصل الواضح بين منطق الخادم والواجهة في تصميم معماري مرن وقابل للتوسيعة.

أظهر المشروع أهمية استخدام الأدوات الحديثة مثل **ANTLR** في بناء المترجمات، كما أبرز دور المفاهيم الأساسية مثل **Visitor** ، **AST** ، **Symbol Table** في تنظيم الكود، تحسين قابليته للصيانة، واكتشاف الأخطاء في مراحل مبكرة. إضافةً إلى ذلك، وفر هذا العمل أساساً قوياً يمكن البناء عليه مستقبلاً لتطوير مزايا متقدمة مثل التحقق الدلالي الكامل، تحسين الأداء، أو حتى تنفيذ الكود.

وبذلك، يحقق هذا المشروع هدفه الأساسي في تقديم نموذج عملٍي ومتكملاً لمترجم متعدد اللغات مخصص لتطبيقات الويب، ويُعد خطوة مهمة نحو فهم أعمق لآليات عمل اللغات البرمجية وبنية الأنظمة الحديثة.