

Python Lex and Yacc

Prepared by

Zeyad moneer abo El-Zahab

Agenda


- ❑ Ambiguous Grammars
- ❑ Embedded Actions
- ❑ Syntax Error Handling
- ❑ PLY Internals
 - Grammer Class
 - Productions
 - LR Items
 - LR Table
 - LR Parser



Ambiguous Grammars

- Ambiguity in grammar occurs when there is **no clear rule on operator precedence**.
- Example: In ' $3 * 4 + 5$ ', should it be ' $(3 * 4) + 5$ ' or ' $3 * (4 + 5)$ '?
- Ambiguity leads to conflicts in parsing:
 - Shift/Reduce Conflict:
 - Occurs when the parser can't decide to shift or reduce.
 - Reduce/Reduce Conflict:
 - Occurs when multiple rules match the input symbols, causing ambiguity.

conflicts in parsing



1	Step	Symbol	Stack	Input Tokens	Action
2	-----	-----	-----	-----	-----
3	1	\$		3 * 4 + 5\$	Shift 3
4	2	\$ 3		* 4 + 5\$	Reduce : expression : NUMBER
5	3	\$ expr		* 4 + 5\$	Shift *
6	4	\$ expr *		4 + 5\$	Shift 4
7	5	\$ expr * 4		+ 5\$	Reduce: expression : NUMBER
8	6	\$ expr * expr		+ 5\$	SHIFT/REDUCE CONFLICT ????
9					
10					

Resolving Ambiguity with Precedence Rules

- Define precedence and associativity for operators:
 - LEFT associativity (e.g., for '+' and '-')
 - RIGHT associativity (e.g., for unary operations like '-')
- Special cases like unary minus can be handled with fictitious tokens:
 - Assign 'UMINUS' a higher precedence for expressions like '-5'
 - Helps override default precedence when necessary.

```
precedence = (  
    ('left', 'PLUS', 'MINUS'),      # الجمع والطرح: اتجاه يساري  
    ('left', 'TIMES', 'DIVIDE'),    # الضرب والقسمة: اتجاه يساري وأولوية أعلى من الجمع والطرح  
    ('right', 'UMINUS'),            # السالب الأحادي: اتجاه يميني وأولوية أعلى  
)
```

Embedded Actions

- Allows code execution at specific points during parsing instead of waiting for full rule parsing.
- Achieved by using empty rules as 'Embedded Actions'.

```
1 def p_foo(p):
2     "foo : A seen_A B C D"
3     print("Parsed a foo", p[1], p[3], p[4], p[5])
4     print("seen_A returned", p[2])
5
6 def p_seen_A(p):
7     "seen_A :"
8     print("Saw an A =", p[-1]) # الوصول إلى الرمز السابق p[-1] يستخدم
9     p[0] = some_value         # تعيين قيمة للقاعدة الفارغة
```

Challenges with Embedded Actions

- Adding embedded actions can create shift-reduce conflicts.
- Example: If the same symbol appears in multiple rules, the parser may not know whether to shift or reduce.
- Solution: Careful design of embedded actions to avoid ambiguities.



```
1  def p_foo(p):
2      """foo : abcd
3          | abcx"""
4
5  def p_abcd(p):
6      "abcd : A B C D"
7
8  def p_abcx(p):
9      "abcx : A B seen_AB C X"
10
11 def p_seen_AB(p):
12     "seen_AB :"
```

Practical Applications: Example of Variable Scoping

- If you are analyzing code in a language like C, you can add an inline rule to control the local scope of variables when entering a block of instructions enclosed between { and }.

```
1  def p_statements_block(p):
2      "statements : LBACE new_scope statements RBACE"
3      # كود الإجراء
4      ...
5      pop_scope() # العودة للنطاق السابق
6
7  def p_new_scope(p):
8      "new_scope :"
9      # إنشاء نطاق جديد للمتغيرات المحلية
10     s = new_scope()
11     push_scope(s)
12     ...
13
```


Syntax Error Handling

- Necessary for user-friendly, robust parsers in production.
- Prevents halting on the first error—continues parsing to identify all issues.
- Standard in production-grade compilers (C, C++, Java).
- A parsing tool for Python similar to Unix yacc.
- Allows defining custom error-handling behaviors with functions like `p_error()`.
- Incorporates recovery mechanisms to resume parsing post-error.

Syntax Error Detection

- Immediate Error Detection
- When a syntax error occurs, PLY immediately detects and calls the error handler.
- `p_error()` function is triggered with the error token as its argument.
- Key Point: No additional tokens are read until the error is addressed.

Steps in Error Handling Process

1. Invoke `p_error()` with offending token (or None if EOF).
2. Enter Recovery Mode to skip calling `p_error()` until 3 tokens are shifted onto the stack.
3. Replace Lookahead Token with an error token if no action is taken.
4. Delete Top of Stack if lookahead token remains error.
5. Restart Parsing if the stack is fully unwound.
6. Error Token Handling if grammar accepts error as a token.

Steps in Error Handling Process

```
1
2 # قواعد اللغة
3 def p_statement_print(p):
4     'statement : PRINT LPAREN ID RPAREN SEMI'
5     print("Print sentence correct")
6
7 # قاعدة للتعامل مع الأخطاء في جملة الطباعة
8 def p_statement_print_error(p):
9     'statement : PRINT error SEMI'
10    print("Grammar error in print sentence. Skip the wrong expression to;")
11
12 # قاعدة عامة للتعامل مع الأخطاء
13 def p_error(p):
14     if not p:
15         print("Grammar error: We have reached the end of the file.")
16         return
17
18     print(f"Grammar error at the symbol: {p.type}")
19
20     # تجاهل الرموز حتى الوصول إلى الفاصلة المنقوطة
21     while True:
22         tok = parser.token() # الحصول على الرمز التالي
23         if not tok or tok.type == 'SEMI':
24             break
25     parser.errok() # إعادة تهيئة وضع الخطأ لاستئناف التحليل بعد تجاوز الخطأ
26
```



► PLY Internals



PLY Internals

LRItems

Breaks down grammar rules into manageable parts.

LRTable

Organizes parsing information for efficient access.

LRParser

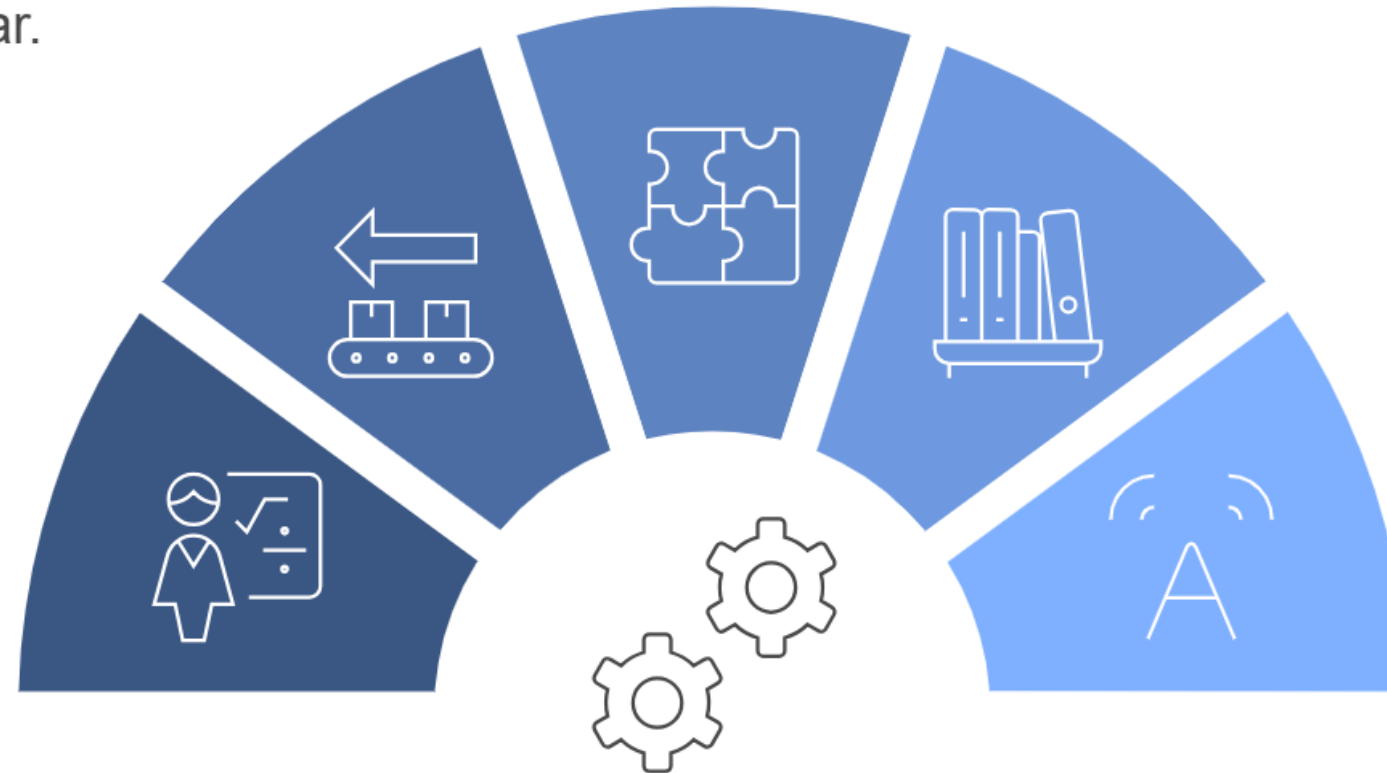
Executes the parsing process using the above components.

Productions

Represents the transformation rules in the grammar.

Grammar Class

Defines the structure and rules of the language.



Grammar Class

1 Terminals

The Grammar Class defines the set of terminal symbols, which correspond to the basic building blocks of the language, such as keywords, identifiers, and punctuation.

2 Non-Terminals

The Grammar Class also defines the set of non-terminal symbols, which represent higher-level language constructs that can be composed of one or more terminal or non-terminal symbols.

3 Production Rules

The Grammar Class specifies the production rules that describe how non-terminal symbols can be rewritten in terms of other symbols, forming the basis of the language's formal grammar.

Productions

Structure

Each production in PLY is represented by a Production object, which stores the left-hand side non-terminal symbol, the right-hand side sequence of symbols, and various metadata such as the production's index and precedence.

Matching

The Production class provides methods for matching input against the production's right-hand side, allowing the parser to determine whether a given sequence of symbols can be reduced according to that production.

Actions

Productions can also be associated with action functions, which are executed when the production is used during parsing to perform tasks such as constructing abstract syntax trees or performing semantic analysis.

Productions



```
1  # A -> B C تعريف قاعدة نحوية مثل
2  production_example = Production (name="A", prod=("B", "C"), number=1, func=None, prec=0)
3  print(production_example.name) # Output: A
4  print(production_example.prod) # Output: ('B', 'C')
```

هذا الرقم يحدد ترتيب
القاعدة أو فهرسها

هي الدالة التي يتم استدعاؤها عندما يتم
استخدام القاعدة في التحليل، وغالبًا ما تبدأ
باسم `p_` متبوعًا باسم القاعدة.

حدد أولوية القاعدة إذا
كان هناك تعارض
بين القواعد

LR Items

LRItems are data structures that track the parser's state during execution, recording **the current position** within a production and **the set of terminals** and **non-terminals** that can be recognized at that point.



```
1  # A -> B . C مثال على خطوة تحليل ضمن القاعدة
2  lr_item_example = LRItem(name="A", prod=("B", ".", "C"), lr_index=1, lr_after=["C"], lr_before="B", lr_next=None)
3  print(lr_item_example.prod)      # Output: ('B', '.', 'C')
4  print(lr_item_example.lr_after)  # Output: ['C']
```

LR Table

1 Table Structure

The LRTable is a two-dimensional data structure that maps the current parser state and the next input symbol to the appropriate parser action, such as shifting to a new state or reducing a production.

3 Table Optimization

PLY's LRTable implementation includes several optimization techniques, such as compacting the table and handling conflicts, to ensure efficient parsing performance.

2 Table Construction

The LRTable is constructed by analyzing the set of LRItems and their transitions, determining the valid actions for each state and input symbol combination.

4 Table Lookup

During parsing, the LRParser uses the LRTable to determine the appropriate action to take based on the current parser state and the next input symbol.

LR Table

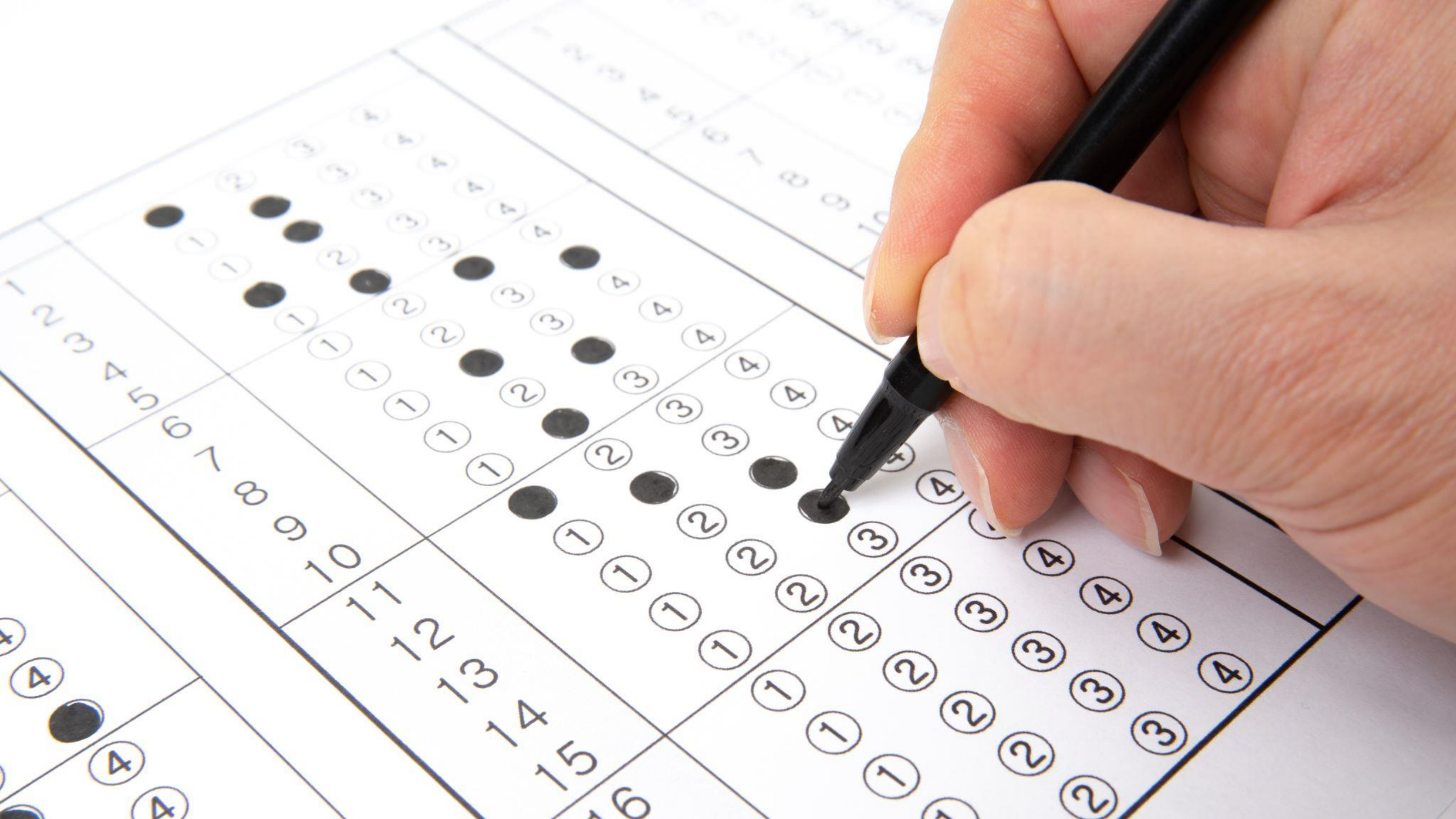
الحالة	(State)	Action (a) / الإجراء	Action (b) / الإجراء	Goto (A) / انتقال	Goto (B) / انتقال
0	S3			1	
1					ACCEPT
2	R2		S4		
3			S5		
4	R3				
5			R1		

Shift إلى الحالة 3.

Reduce باستخدام
الحالة 1.

LR Parser

- The LRParser initializes the parsing process by constructing the LRTable and setting the parser's starting state.
- The parser then repeatedly reads the next input symbol and uses the LRTable to determine the appropriate action, such as shifting to a new state or reducing a production.
- When a reduction is performed, the parser pops the required number of symbols off the parser stack and applies the associated action function to construct the resulting non-terminal symbol
- The parsing process continues until the parser reaches an accept state, indicating that the input has been successfully parsed according to the language's grammar.



Task:

Building a Basic Interpreter With PLY (Python Lex-Yacc)

- Understand how to define and use tokens for lexing text input.
- Implement a lexer that can identify various components of the language.
- Implement a parser that can recognize and evaluate expressions.
- Add support for error handling, debugging, and state management within the lexer and parser.

Thank you

Any questions?