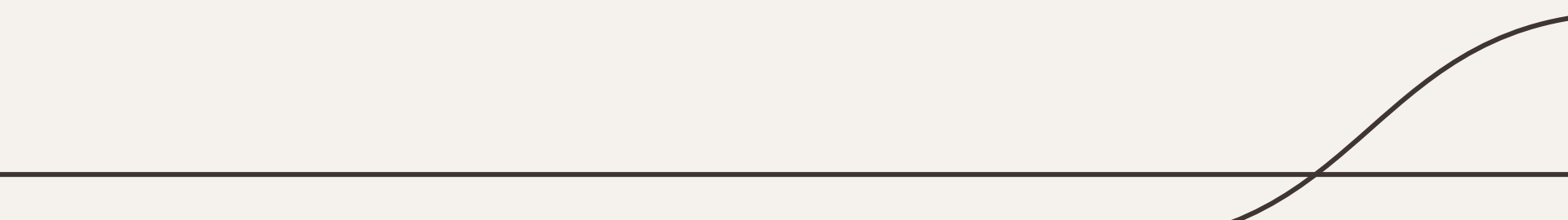




Syntax Analyser

Team Compilyashki



Our team: Compilyashki

Polina Pushkareva

Responsible for
organizational aspects
and report

Matthew Rusakov

Responsible for testing for
syntax analyser coding



Aliia Bogapova

Responsible for testing for
syntax analyser coding

Technologies



Project F

Interpreter for lisp-like
(functional) language



Hand-based parser in Java



Hand-based lexer in Java

Members Contribution

Polina Pushkareva

- Documentation Structuring & Design
- Presentation Drafting & Design



Matthew Rusakov


- Bug Fixes & Code Refinements
- Object-Oriented Design & Refactoring
- Core Codebase Development
- Unit Testing

Aliia Bogapova

- Core Parser Logic Development
- Abstract Syntax Tree (AST) Generation


A couple of Example Programs in Functional Language

```
3 (plus 3 4)
4 (minus 10 5)
5 (times 6 7)
6 (divide 20 4)
```

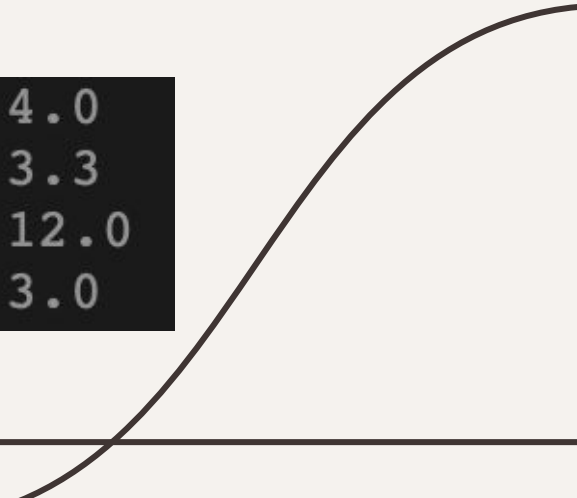


```
7
5
42
5
```

```
3 (plus 1.5 2.5)
4 (minus 5.5 2.2)
5 (times 3.0 4.0)
6 (divide 9.0 3.0)
```



```
4.0
3.3
12.0
3.0
```



A couple of Example Programs in Functional Language

```
3 (cons 1 (cons 2 (cons 3 null)))  
4 (head (cons 1 (cons 2 null)))  
5 (tail (cons 1 (cons 2 null)))
```

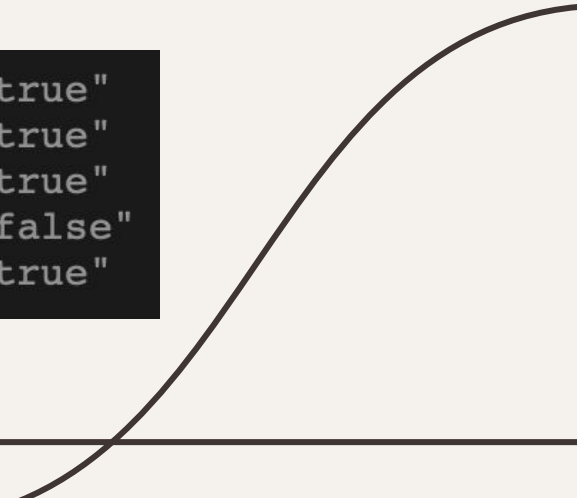


```
(1 2 3)  
1  
(2)
```

```
3 (isint 42)  
4 (isreal 3.14)  
5 (isbool true)  
6 (isnull 1)  
7 (islist (cons 1 (cons 2 null)))
```



```
"true"  
"true"  
"true"  
"false"  
"true"
```



A couple of Example Programs in Functional Language

```
3 (equal 5 5)
4 (nonequal 5 4)
5 (less 3 5)
6 (lesseq 5 5)
7 (greater 6 5)
8 (greatereq 5 5)
```



```
"true"
"true"
"true"
"true"
"true"
"true"
```

```
3 (and true false)
4 (or true false)
5 (xor true false)
6 (not true)
```



```
"false"
"true"
"true"
"false"
```

A couple of Example Programs in Functional Language

```
3 (func square (x) (times x x))  
4 (square 4)  
5  
6 (func sumOfSquares (a b) (plus (square a) (square b)))  
7 (sumOfSquares 3 4)
```



16
25

```
3 (setq increment (lambda (x) (plus x 1)))  
4 (increment 7)  
5  
6 (setq applyTwice (lambda (f x) (f (f x)))) ;  
7 (applyTwice increment 5)
```



8
7

A couple of Example Programs in Functional Language

```
3 (cond (less 3 5) (plus 2 2) (minus 5 3))  
4 (cond (greater 5 3) (plus 10 5) (minus 5 3))
```



4
15

```
3 (setq counter 0)  
4 (while (less counter 5)  
5   (setq counter (plus counter 1)))
```

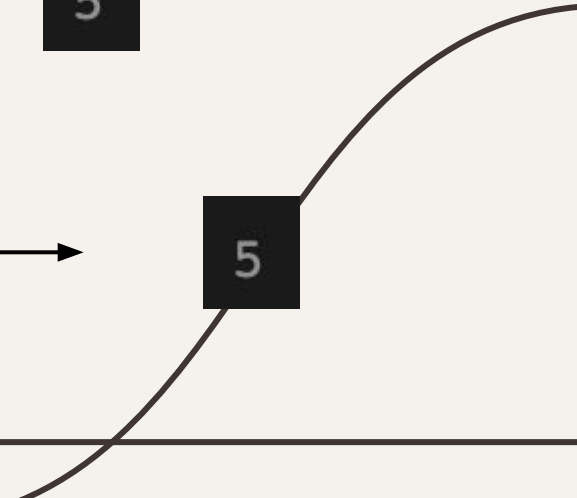


5

```
3 (setq counter 0)  
4 (while (less counter 10)  
5   (cond (equal counter 5)  
6         (break)  
7         (setq counter (plus counter 1))))
```



5



A couple of Example Programs in Functional Language

```
3 (func triple (x) (times x 3))  
4 (func addAndTriple (a b) (triple (plus a b)))  
5 (addAndTriple 2 3)
```



15

```
3 (setq code (quote (plus 7 8)))  
4 (eval code)
```



15



Description of the Implementation



Our parser uses **top-down parsing** to construct the **Abstract Syntax Tree (AST)**, representing the program structure. Each node in the AST corresponds to language constructs, such as expressions or functions, with high-order functions handled as composite nodes. A **symbol table** is maintained to track entities, storing details like the entity's name, value, span, and additional metadata such as line numbers for error reporting and debugging.

Description of the Implementation

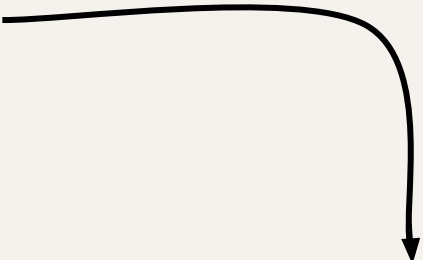


We applied **Object-Oriented Programming (OOP) principles** to ensure maintainability and scalability. Key design patterns include:

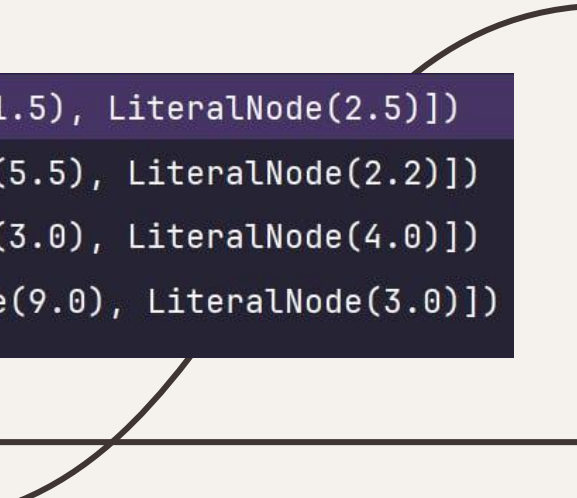
- **Singleton** for large, stateful components like the Lexer, Parser, and Factory, ensuring only one instance is active.
- **Factory Pattern** for node creation, streamlining the generation of various AST node types.
- **Visitor Pattern** for tree traversal and exporting ASTs to text files, allowing flexible output formats.
- **Composite Pattern** to organize AST nodes, particularly for complex structures like high-order functions.

A Few Examples of Work

```
(plus 1.5 2.5)  
(minus 5.5 2.2)  
(times 3.0 4.0)  
(divide 9.0 3.0)
```

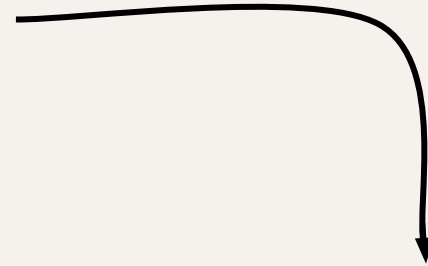


```
OperationNode(operator=plus, operands=[LiteralNode(1.5), LiteralNode(2.5)])  
OperationNode(operator=minus, operands=[LiteralNode(5.5), LiteralNode(2.2)])  
OperationNode(operator=times, operands=[LiteralNode(3.0), LiteralNode(4.0)])  
OperationNode(operator=divide, operands=[LiteralNode(9.0), LiteralNode(3.0)])
```



A Few Examples of Work


```
(cons 1 (cons 2 (cons 3 (1 2 3))))  
(head (cons 1 (cons 2 ())) )  
(tail (cons 1 (cons 2 (3 4))))
```




```
ConsNode(head=LiteralNode(1), tail=ConsNode(head=LiteralNode(2), tail=ConsNode(head=LiteralNode(3), tail=ListNode(elements=[LiteralNode(1),LiteralNode(2),Lit  
HeadNode(list=ConsNode(head=LiteralNode(1), tail=ConsNode(head=LiteralNode(2), tail=ListNode(elements=[]))))))  
TailNode(list=ConsNode(head=LiteralNode(1), tail=ConsNode(head=LiteralNode(2), tail=ListNode(elements=[LiteralNode(3),LiteralNode(4)]))))))
```

A Few Examples of Work


```
(isint 42)  
(isreal 3.14)  
(isbool true)  
(isnull 1)  
(islist (cons 1 (cons 2 ())))
```



```
PredicateNode(predicate=isint, element=LiteralNode(42))  
PredicateNode(predicate=isreal, element=LiteralNode(3.14))  
PredicateNode(predicate=isbool, element=LiteralNode(true))  
PredicateNode(predicate=isnull, element=LiteralNode(1))  
PredicateNode(predicate=islist, element=ConsNode(head=LiteralNode(1), tail=ConsNode(head=LiteralNode(2), tail=ListNode(elements=[]))))
```



A Few Examples of Work



```
(equal 5 5)  
(nonequal 5 4)  
(less 3 5)  
(lesseq 5 5)  
(greater 6 5)  
(greatereq 5 5)
```

```
ComparisonNode(comparison=equal, leftElement=LiteralNode(5), rightElement=LiteralNode(5))  
ComparisonNode(comparison=nonequal, leftElement=LiteralNode(5), rightElement=LiteralNode(4))  
ComparisonNode(comparison=less, leftElement=LiteralNode(3), rightElement=LiteralNode(5))  
ComparisonNode(comparison=lesseq, leftElement=LiteralNode(5), rightElement=LiteralNode(5))  
ComparisonNode(comparison=greater, leftElement=LiteralNode(6), rightElement=LiteralNode(5))  
ComparisonNode(comparison=greatereq, leftElement=LiteralNode(5), rightElement=LiteralNode(5))
```


A Few Examples of Work



```
(setq double (lambda (x) (times x 2)))  
(double 4)  
  
(setq increment (lambda (x) (plus x 1)))  
(increment 7)  
  
(setq applyTwice (lambda (f x) (f (f x))))  
(applyTwice increment 5)  
  
(setq subtractTwo (lambda (x) (minus x 2)))  
(applyTwice subtractTwo 10)
```

```
AssignmentNode(variable=double, value=LambdaNode(parameters=[x], body=OperationNode(operator=times, operands=[AtomNode(x), LiteralNode(2)])))  
FunctionCallNode(functionName=double, parameters=[AtomNode(double), LiteralNode(4)])  
AssignmentNode(variable=increment, value=LambdaNode(parameters=[x], body=OperationNode(operator=plus, operands=[AtomNode(x), LiteralNode(1)])))  
FunctionCallNode(functionName=increment, parameters=[AtomNode(increment), LiteralNode(7)])  
AssignmentNode(variable=applyTwice, value=LambdaNode(parameters=[f, x], body=ListNode(elements=[AtomNode(f), ListNode(elements=[AtomNode(f), AtomNode(x)])])))  
FunctionCallNode(functionName=applyTwice, parameters=[AtomNode(applyTwice), AtomNode(increment), LiteralNode(5)])  
AssignmentNode(variable=subtractTwo, value=LambdaNode(parameters=[x], body=OperationNode(operator=minus, operands=[AtomNode(x), LiteralNode(2)])))  
FunctionCallNode(functionName=applyTwice, parameters=[AtomNode(applyTwice), AtomNode(subtractTwo), LiteralNode(10)])
```

The image features a light gray background with dark gray wavy lines in the corners, creating a decorative frame. The text "Thanks for attention!" is centered in a large, bold, black serif font.

**Thanks for
attention!**

Link to GitHub:

https://github.com/MattWay224/F24CompilerConstruction_Compilyashki