



Parser & SDT:

A Technical Implementation of an LL(1) Analyzer

An Intelligent, Visual, and Robust Analyzer

Team:

320206102

316255819

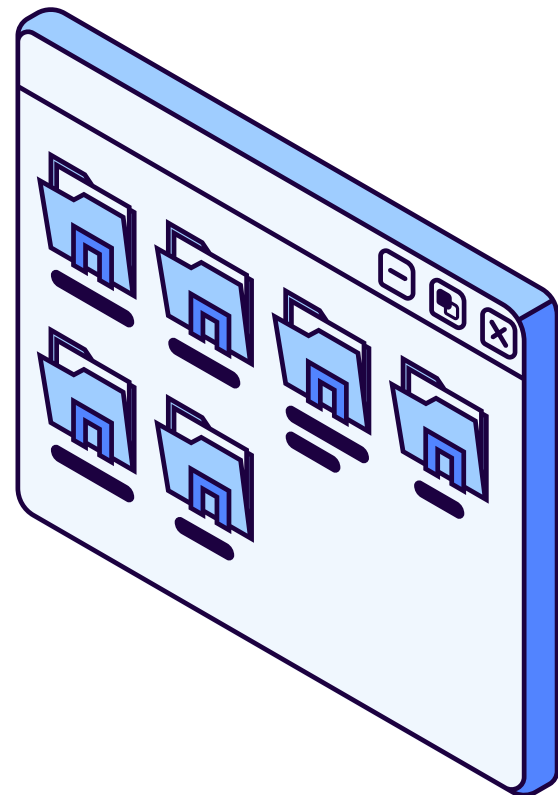
423031180

320117174

320340312

Core Features

- Full Graphical User Interface (GUI).
- File Loading & Pre-loaded Examples.
- LL(1) Recursive Descent Parser.
- Support for arithmetic expressions.
- Real-time semantic validation via SDT.
- Differentiated Error Reporting (Syntactic vs. Semantic).
- Automated AST (Abstract Syntax Tree) generation via Graphviz.



The Grammar: Our "Blueprint"

A Context-Free Grammar implemented via recursive function

Key Productions:

- $\text{Program} \rightarrow \text{Statement}^+$
- $\text{Statement} \rightarrow \text{Declaration} \mid \text{Assignment}$
- $\text{Declaration} \rightarrow (\text{const})? \text{Type ID} = \text{Expr} ;$
- $\text{Assignment} \rightarrow \text{ID} = \text{Expr} ;$
- $\text{Expr} \rightarrow \text{Term Expr}'$
- $\text{Expr}' \rightarrow + \text{Term Expr}' \mid - \text{Term Expr}' \mid \varepsilon$
- $\text{Term} \rightarrow \text{Factor Term}'$
- $\text{Term}' \rightarrow * \text{Factor Term}' \mid / \text{Factor Term}' \mid \varepsilon$
- $\text{Factor} \rightarrow (\text{Expr}) \mid \text{ID} \mid \text{NUM} \mid \text{CHAR} \mid \dots$



The Parser: The "Engine"

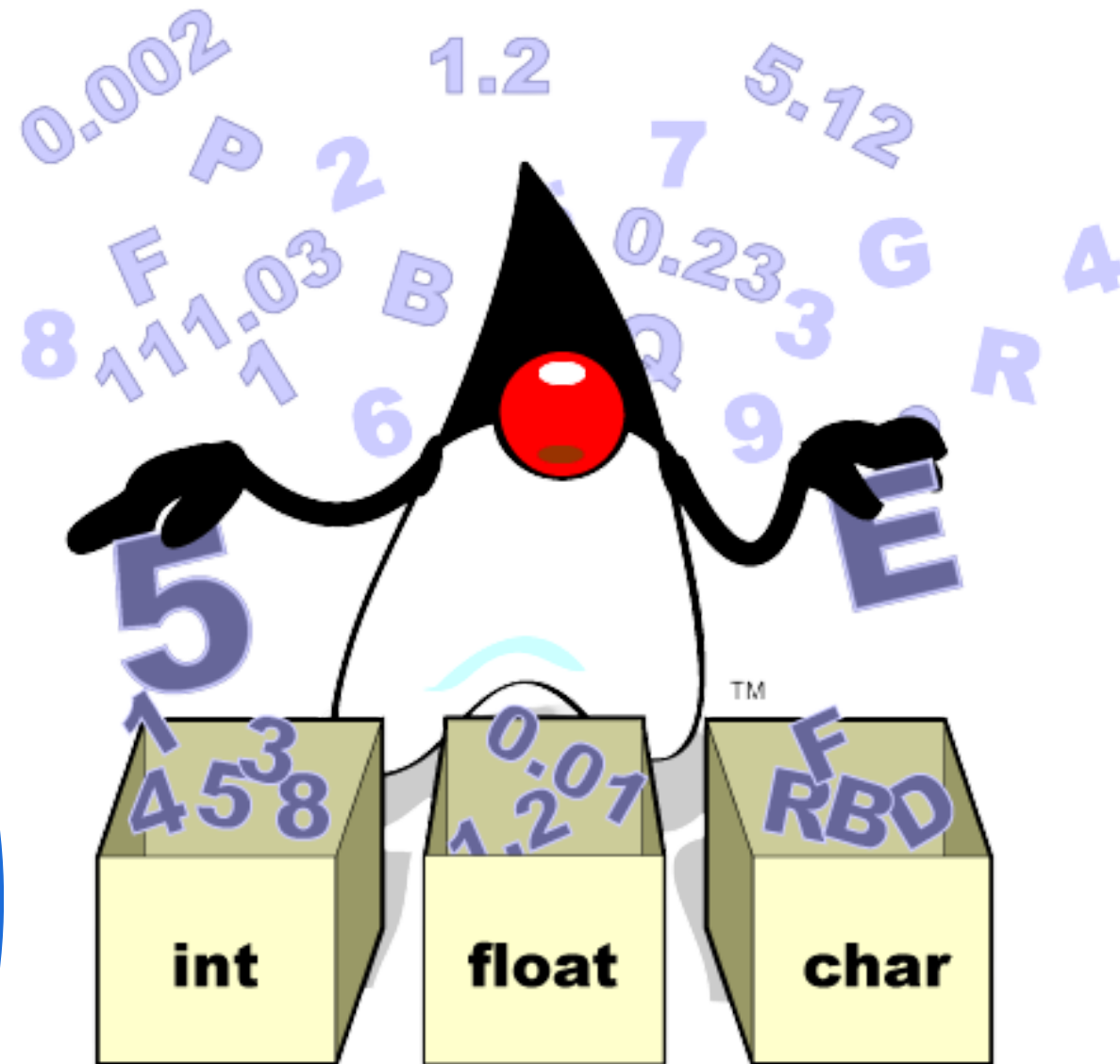
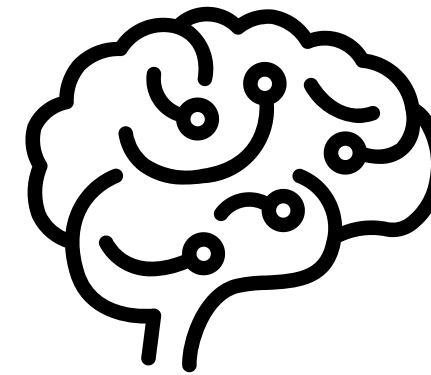
How the grammar is implemented

How it works:

- Each non-terminal (e.g., Program, Expr, Term) is a Python function.
- The parser uses one token of lookahead to predict which function to call.
- Functions call each other recursively, "descending" through the grammar to match the token stream.



The Brain: The Symbol Table



Schema (What it stores):

- Name: The variable name (e.g., x).
- Type: The data type (e.g., int).
- Value: The current value (e.g., 10)

A stateful data structure for semantic analysis.

The Action: Syntax-Directed Translation (SDT)

SDT actions are code snippets embedded directly into the recursive-descent functions.

Trace 1 (Writing): `int x = 10;`

1. `declaration()` function parses `int`, `x`, `=`.
2. `declaration()` calls `expr()` to get the value `10`.
3. SDT ACTION: `self.symbol_table.add('x', 'int', 10, False)` is executed.
4. Log output: SDT: Declared 'x' (int) = 10.

Trace 2 (Reading / Error): `y + 5`

1. `expr()` calls `term()`, which calls `factor()`.
2. `factor()` sees the ID token `y`.
3. SDT ACTION: `symbol = self.symbol_table.get('y')` is executed.
4. `symbol` is `None`. The action throws a `SemanticError`.

RESULT 1

The Control Center

Key Fetures

- Clean Input / Output areas.
- File Loading & Examples.
- Multi-View Tabbed Results:
 - Output (The Log)
 - Tree (Text)
 - Symbols (The "Brain" activity)



RESULT 2

The "Happy Path"

Key Points:

- Status: Parsing Success! | SDT Verified!
- Lexer: Correctly tokenized the input.
- Parser: Matched the Declaration grammar rule.
- SDT: Fired the semantic action: SDT:
Declared 'x' (int) = 10

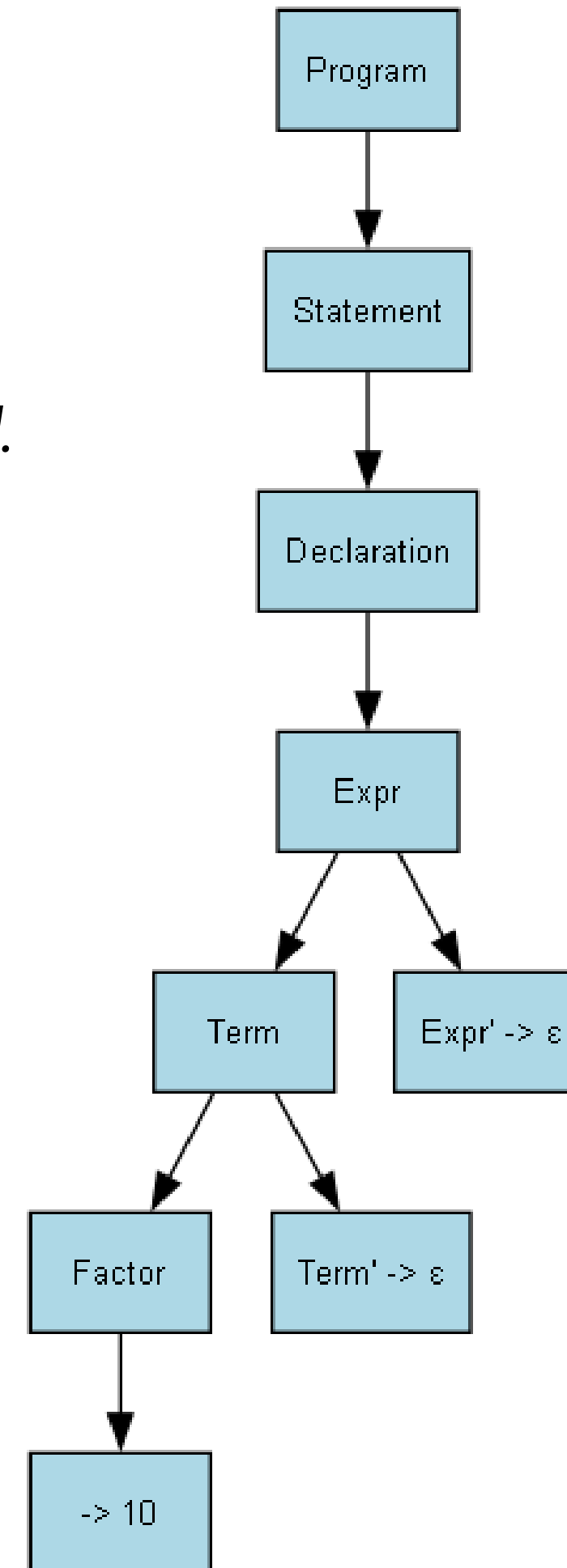
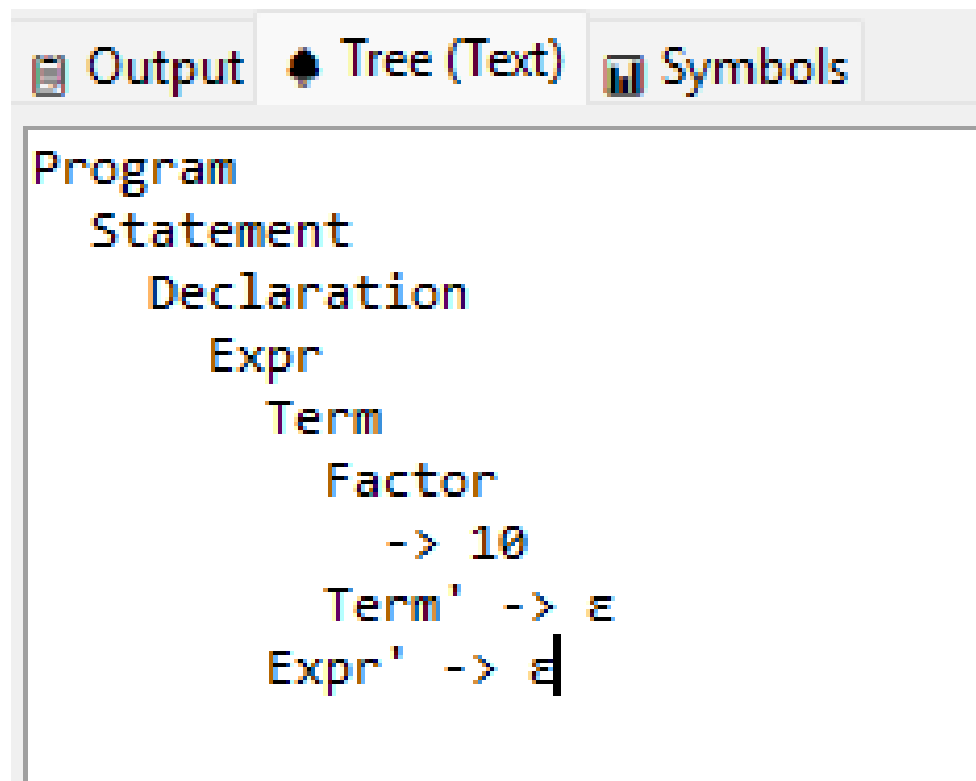
```
✓ Parsing Success! | SDT Verified!
```

```
Output Tree (Text) Symbols
=====
LEXICAL ANALYSIS
=====
0: <KEYWORD, 'int'>
1: <ID, 'x'>
2: <OP, '='>
3: <NUM, '10'>
4: <SPECIAL, ';'>
5: <EOF, '$'>
=====
SYNTACTIC AND SEMANTIC (SDT) ANALYSIS
=====
Match: int (KEYWORD)
Match: x (ID)
Match: = (OP)
Match: 10 (NUM)
Match: ; (SPECIAL)
SDT: Declared 'x' (int) = 10
✓ Parsing Success!
✓ SDT Verified!
```

RESULT 3

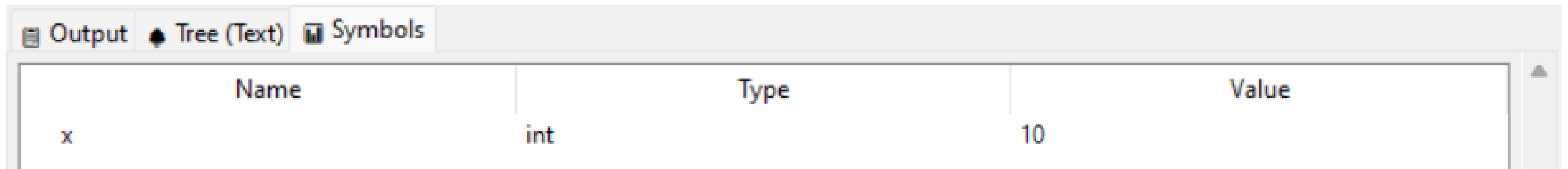
The Visual Proof

We don't just say it parsed, we show you how it parsed.



RESULT 4

The Semantic Proof (The Symbol Table)



The screenshot shows a window titled 'Symbols' with a table containing one entry. The table has four columns: 'Name', 'Type', and 'Value'. The entry for 'x' is of type 'int' and has a value of '10'.

Name	Type	Value
x	int	10

This proves the SDT action worked.

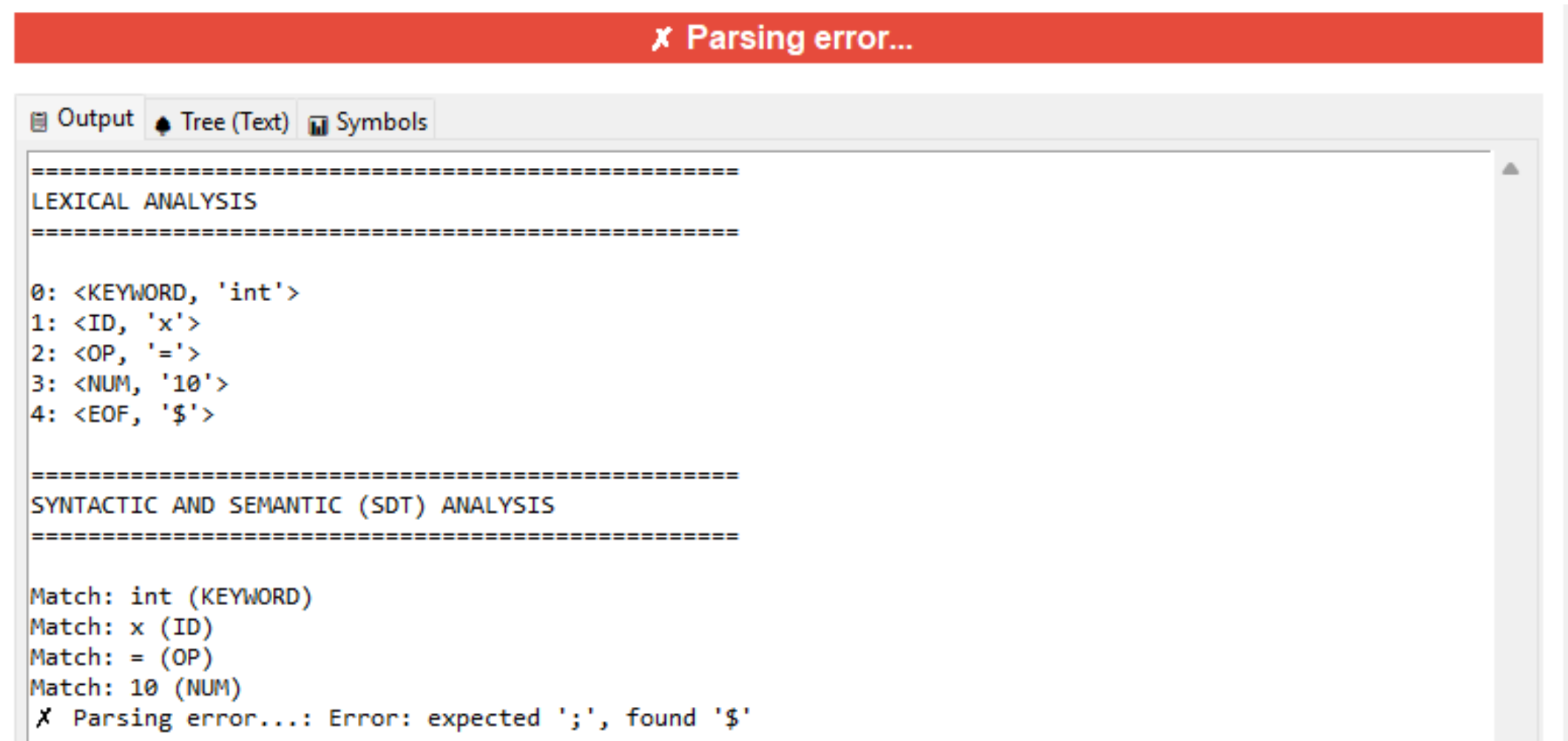
The parser didn't just see 'x', it

understood 'x'.

RESULT 5

Robustness (Handling Syntactic Errors)

- Input: int x = 10 (Missing ;)
- Status: Parsing error...



The screenshot shows a software interface with a red header bar at the top containing a red 'X' icon and the text 'Parsing error...'. Below this is a tabbed window with three tabs: 'Output' (selected), 'Tree (Text)', and 'Symbols'. The 'Output' tab displays the following text:

```
=====
LEXICAL ANALYSIS
=====

0: <KEYWORD, 'int'>
1: <ID, 'x'>
2: <OP, '='>
3: <NUM, '10'>
4: <EOF, '$'>

=====
SYNTACTIC AND SEMANTIC (SDT) ANALYSIS
=====

Match: int (KEYWORD)
Match: x (ID)
Match: = (OP)
Match: 10 (NUM)
X Parsing error....: Error: expected ';', found '$'
```

The parser stops and gives a precise error: Error: expected ';', found '\$'.

RESULT 6

The "Smart" Error - Catching Semantic (SDT) Errors

✓ Parsing Success! | ✗ SDT error...

```
=====
SYNTACTIC AND SEMANTIC (SDT) ANALYSIS
=====
```

```
Match: int (KEYWORD)
Match: x (ID)
Match: = (OP)
Match: 10 (NUM)
Match: ; (SPECIAL)
SDT: Declared 'x' (int) = 10
Match: x (ID)
Match: = (OP)
Match: y (ID)
✓ Parsing Success!
✗ SDT error...: Variable 'y' not declared
```

```
✓ Tree image generated and opened! (parser_ast.png)
```

- Input: `int x = 10; x = y + 5;`
- Status: Parsing Success! | ✗ SDT error...

This is our most important feature

Syntax: The code is grammatically perfect.
The parser succeeded.

Semantics: The SDT action tried to get *y* from the Symbol Table, but it failed.

Result: SDT error... Variable 'y' not declared.

RESULT 7

Visualizing Despite Errors

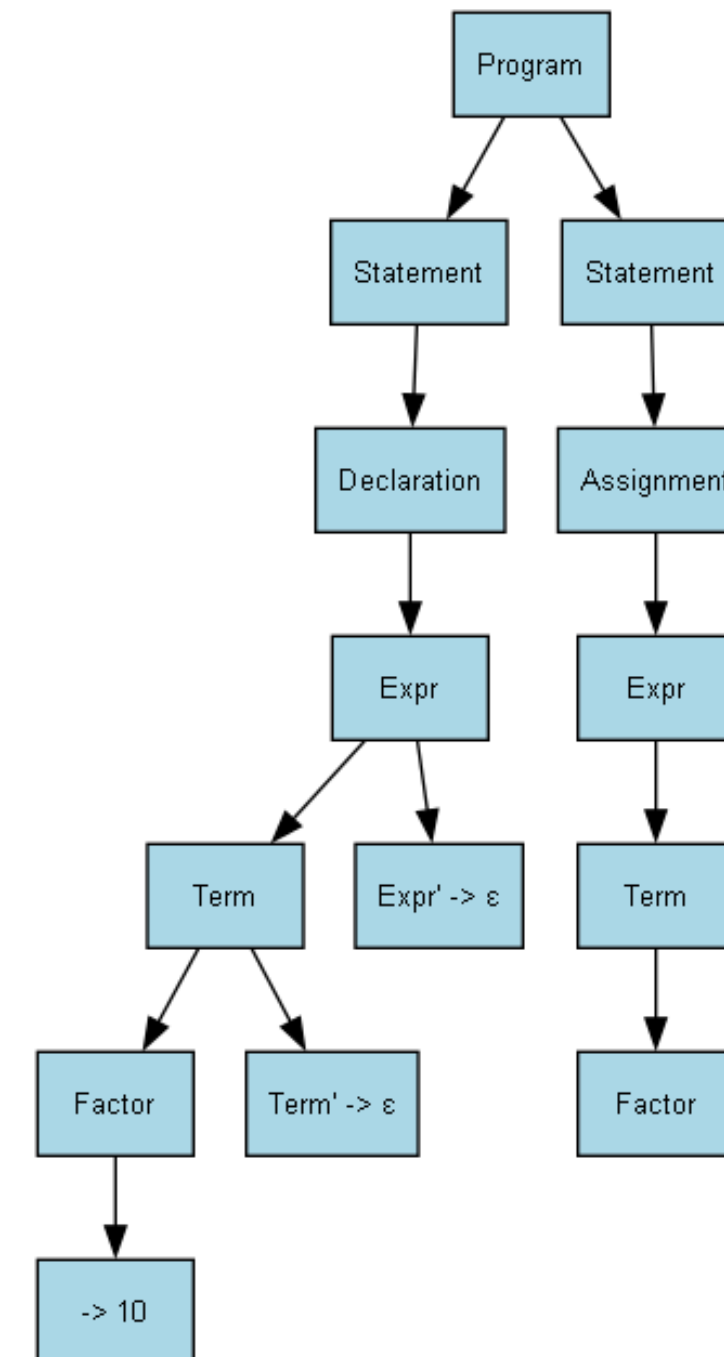
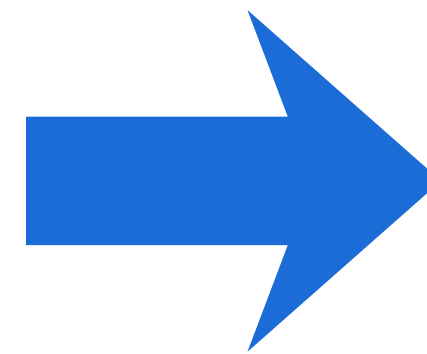
When an SDT Error occurs, we still generate the AST.

✓ Parsing Success! | ✗ SDT error...

SYNTACTIC AND SEMANTIC (SDT) ANALYSIS

```
=====
Match: int (KEYWORD)
Match: x (ID)
Match: = (OP)
Match: 10 (NUM)
Match: ; (SPECIAL)
SDT: Declared 'x' (int) = 10
Match: x (ID)
Match: = (OP)
Match: y (ID)
✓ Parsing Success!
✗ SDT error...: Variable 'y' not declared
```

✓ Tree image generated and opened! (parser_ast.png)



RESULT 8

Power - Handling Complex Expressions

✓ Parsing Success! | SDT Verified!

Output Tree (Text) Symbols

LEXICAL ANALYSIS

```
0: <KEYWORD, 'int'>
1: <ID, 'value'>
2: <OP, '='>
3: <SPECIAL, '('>
4: <NUM, '10'>
5: <OP, '+'>
6: <NUM, '5'>
7: <SPECIAL, ')'>
8: <OP, '*'>
9: <NUM, '2'>
10: <OP, '-'>
11: <NUM, '8'>
12: <OP, '/'>
13: <NUM, '4'>
14: <SPECIAL, ';'>
15: <EOF, '$'>
```

SYNTACTIC AND SEMANTIC (SDT) ANALYSIS

```
Match: int (KEYWORD)
Match: value (ID)
Match: = (OP)
Match: ( (SPECIAL)
Match: 10 (NUM)
Match: + (OP)
Match: 5 (NUM)
SDT: 10 + 5 = 15
Match: ) (SPECIAL)
Match: * (OP)
Match: 2 (NUM)
SDT: 15 * 2 = 30
Match: - (OP)
Match: 8 (NUM)
Match: / (OP)
Match: 4 (NUM)
SDT: 8 / 4 = 2.0
SDT: 30 - 2.0 = 28.0
Match: ; (SPECIAL)
SDT: Declared 'value' (int) = 28.0
✓ Parsing Success!
✓ SDT Verified!
```

- Input: $\text{int value} = (10 + 5) * 2 - 8 / 4;$
- Parser correctly applies precedence and parenthesis rules.
- Calculated Result: 28.
- The AST visually confirms that $*$ and $/$ are deeper than $-$.



Conclusion

Objective Met

We built a functional, robust LL(1) parser.

Visual Validation

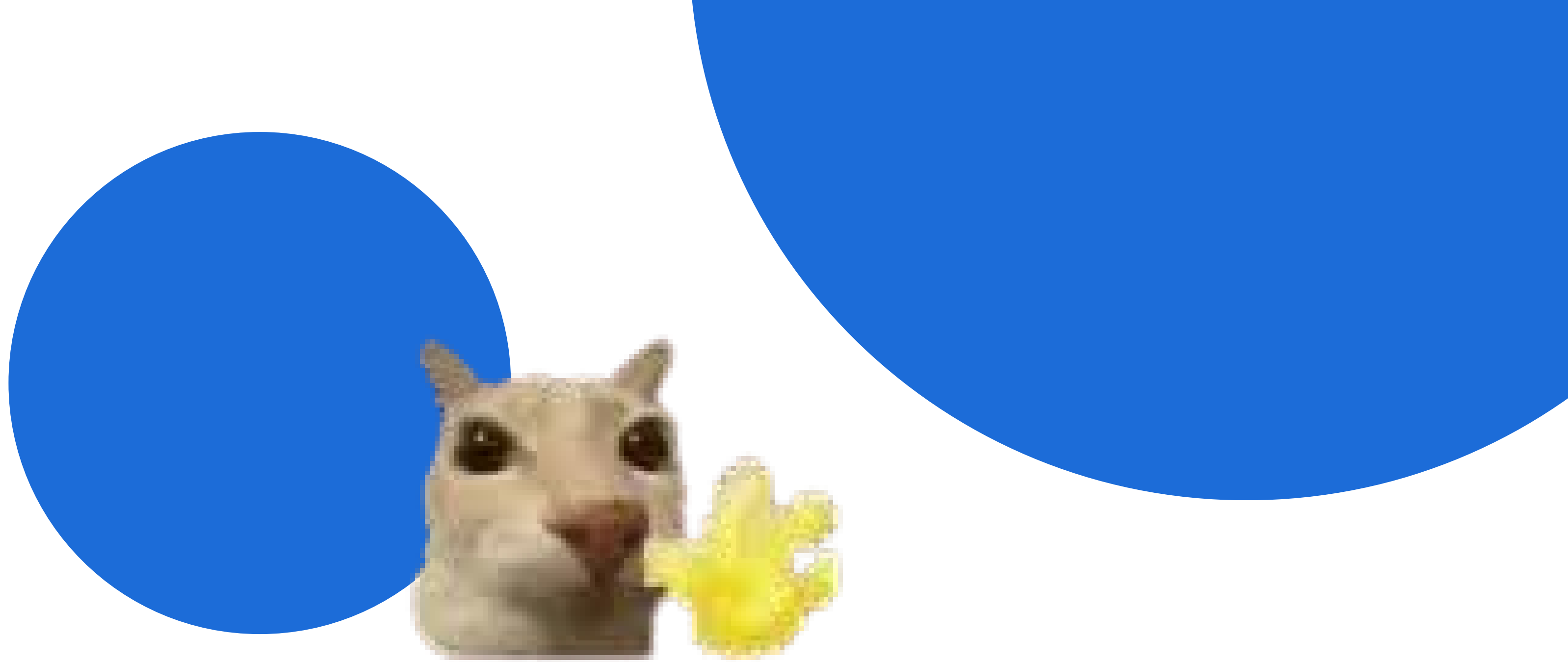
The AST provides clear, graphical proof of parsing.

Smart Feedback

Successfully distinguished Syntactic vs. Semantic errors.

Intelligent Core

The Symbol Table and SDT actions create a truly "smart" analyzer.



THANK YOU