Compilation Techniques

School of Computing and Creative Arts

Class L5AC

Lecturer: **Ir. TRI ASIH BUDIONO, M.I.T.**

Members:

Albertus Santoso - 2702334885

Gabriel Anderson - 2702256315

Justin Hadinata - 2702298236

Introduction

Background

Designing user interfaces is a fundamental part of modern software development, and yet most UI frameworks involve steep learning curves, verbose syntax, or large dependency counts. For starters, the gap between a layout idea and the code required to render it can be unnecessarily huge. Traditional UI toolkits often mix styling, logic, and layout rules in ways that make experimentation slow and confusing.

This project investigates if a minimal, CSS-like language could make the creation of 2D layouts simpler, more readable, and more accessible. The idea behind SimpleUI is to describe shapes and layouts using concise statements, then compile them into a fully runnable output. In this case, portable Python/Turtle code. That's interesting because it shows how even a small DSL can automate repetitive tasks, enforce structure, and offer immediate visual feedback.

This project is important from a learning perspective, as it covers the full compiler pipeline: tokenizing, parsing, AST construction, code generation, and interactive rendering. It allows us to understand how languages are designed from scratch, how grammars translate into syntax rules, and how high-level descriptions can be transformed into executable programs. Building a full, end-to-end compiler for even a small DSL is an exercise that will strengthen our understanding of programming languages, abstractions, and software tooling.

Problem description

Creating simple 2D UI layouts often requires tools that are unnecessarily complex for beginners or for quick prototyping. Existing UI frameworks demand large amounts of boilerplate code, an understanding of layout engines, or reliance on heavy graphical editors, making even basic shapes or mockups harder to implement than they should be. There is no lightweight, readable way to describe layouts using short syntax and instantly transform them into visual output without external dependencies. This creates a gap between the simplicity of the idea and the complexity of implementing it in code. The problem this project addresses is how to design a minimal, beginner-friendly domain-specific language that can describe UI layouts concisely, and how to build a complete compiler that converts this language into executable Python/Turtle scripts. The challenge involves creating a full compiler pipeline such as lexing, parsing, AST construction, and code generation while keeping the language intuitive, the implementation lightweight, and the output portable.

Related Work

A closely related system is Protovis, a declarative visualization toolkit that allows designers to specify graphical marks and layouts using a concise, domain-specific language rather than imperative drawing code. Protovis demonstrates how separating what is drawn from how it is rendered can dramatically simplify the creation of visual output, reduce boilerplate, and make graphical tools more accessible to beginners. Although Protovis focuses on data-driven visualizations while SimpleUI targets 2D UI layout prototypes, both share the same fundamental design philosophy: using a lightweight, human-readable DSL to bridge the gap between conceptual layout ideas and concrete rendered output. This connection validates the approach taken in SimpleUI, showing that expressive declarative languages have a long-standing role in making visual programming tasks easier and more intuitive.
Ref: 2010-Protovis-InfoVis.pdf

Another related work is Goderis and Deridder's work presents a declarative domain-specific language (DSL) for user interface specification that separates UI concerns (layout, component structure, interaction) from application logic by expressing UI components and layout rules as declarative "facts and rules," which are processed by a problem solver to generate concrete UI on a target platform. This separation-of-concerns approach and the use of a lightweight, meta-programming DSL to describe UI mirrors the core philosophy of SimpleUI: rather than writing imperative drawing code, developers declare UI structure and layout in a small, readable language, which is then compiled into runnable code (in your case, Python/Turtle). While the 2004 framework targets more general GUI components and platform independence (e.g. allowing reuse or re-targeting across devices), and uses logic programming under the hood, the fundamental insight remains the same: a minimal, declarative description of UI can dramatically simplify UI creation, reduce boilerplate, and improve maintainability. By situating SimpleUI alongside this prior DSL-based approach, we highlight that SimpleUI continues a tradition of using declarative languages to treat UI as a first-class, language-level abstraction, validating the design goals of simplicity, readability, and portability.
Ref: vub-prog-tr-04-05.pdf

Implementation

Formal Description of the Computational Problem

The core objective of this project is to implement a compiler for SimpleUI, a domain-specific language (DSL) used for declarative 2D User Interface layout. The computational problem involves translating high-level, human-readable declarations of shape, position, size, and color into low-level imperative execution commands that run using the Python standard library (specifically Turtle graphics).

The compiler must perform three major tasks:

1. Recognize valid tokens from the source text (lexical analysis).
2. Parse the sequence of tokens into structured shape statements (syntax analysis).
3. Convert the parsed structure into executable Python code (code generation).

Lexical Specification (REGEX)

The lexical analyzer (lexer.py) converts raw source code into a list of tokens. The following Regular Expressions define all token types in SimpleUI.

Keywords (literal strings):

- rectangle
- circle
- line

Attribute Keys:

- left
- top
- w
- h
- fill
- outside
- rounded

Data Types:

- NUMBER: \d+(\.\d+)?
  Matches integers and floating-point numbers.
- HEX_COLOR: #[0-9a-fA-F]{6}
  Matches HTML-style color codes (e.g., #FF00AA).

- UNIT: px
  Matches the pixel unit suffix.

Delimiters:

- COMMA: ,
- SEMICOLON: ;

The lexer also ignores whitespace and comments, producing only meaningful tokens for the parser.

Grammar Specification (EBNF) The syntax of SimpleUI is context-free.

The following grammar describes all valid SimpleUI programs using standard Extended Backus–Naur Form (EBNF). This grammar corresponds directly to the grammar implemented in the file grammar.lark.

Rule Name (Non-Terminal)	Definition	Notes
start	shape +	A SimpleUI program consists of one or more (+) shape statements.
shape	(property ,) * shape_name ;	A shape is a sequence of zero or more (*) properties followed by a shape keyword and terminated by a semicolon.
property	left_pos | top_pos | width | height | fill | outside | rounded	A property must be one of these types.
left_pos	NUMBER "px" "left"	Defines the X-position.
top_pos	NUMBER "px" "top"	Defines the Y-position.
width	"w" : NUMBER "px"	Defines the width attribute.
height	"h" : NUMBER "px"	Defines the height attribute.
fill	"fill" : color	Defines the fill color attribute.
outside	"outside" : color	Defines the border color attribute.
rounded	"rounded"	The optional style attribute for rounded corners.
shape_name	RECTANGLE | CIRCLE | LINE	The final keyword defining the shape type.
color	HEX | NAME	A color can be a hexadecimal code or a recognized color name.
Terminals (Examples)		
NUMBER	/[0-9]+\.?[0-9]*/	Matches integers (e.g., 100) and floats (e.g., 10.5).
HEX	/#[0-9A-Fa-f]{6}/	Matches 6-digit hex color codes (e.g., #FF00AA).
NAME	/ [a-zA-Z]+/	Matches color names (e.g., red, blue).

This grammar ensures that every statement ends with a semicolon and contains a sequence of attributes followed by a shape keyword.

Design of the Compiler

The SimpleUI compiler uses a clean, multi-pass architecture that separates lexing, parsing, AST construction, and code generation to ensure clarity and maintainability.

Component Architecture

1. Lexer (Tokenizer)
   The custom lexer in lexer.py reads characters one by one and groups them into

meaningful tokens. It also tracks precise line and column numbers. This allows the compiler to report detailed, user-friendly error messages when invalid syntax is encountered.

2. Parser and AST Construction

The parser is implemented with the Lark parsing library. It consumes the token sequence according to the grammar in grammar.lark. Instead of returning a generic parse tree, Lark uses a Transformer class to convert nodes directly into a typed Abstract Syntax Tree (AST).

AST nodes are defined in ast_nodes.py using Python dataclasses. Example node types include:

- RectangleNode
- CircleNode
- LineNode

These dataclasses enforce type checking (e.g., width must be a float) and insert default values automatically (e.g., an unspecified “outside” color defaults to transparent).

3. Code Generator

The module code_generator.py traverses the AST and emits Python Turtle commands. It translates declarative UI descriptions into imperative steps.

A key responsibility is **coordinate transformation**. SimpleUI uses a top-left coordinate system similar to typical UI frameworks, while Turtle uses a centered Cartesian coordinate system. The generator converts coordinates accordingly so that shapes appear at the correct positions.

Complexity Analysis

Lexical Analysis

The lexer reads the source text once, character by character.

Time complexity: **O(N)**, where N is the length of the input.

Parsing

The Lark parser uses an efficient algorithm (LALR(1) for this grammar). For a non-ambiguous grammar like SimpleUI’s, parsing is linear relative to the number of tokens.

Time complexity: **O(T)**, where T is the number of tokens.

Code Generation

The code generator processes each AST node once and produces a corresponding block of Turtle commands.

Time complexity: **O(S)**, where S is the number of shape statements.

Overall Complexity

Combining all stages, the compiler runs in overall linear time relative to input size. Final complexity: **O(N)**.

This makes the SimpleUI compiler efficient for typical UI layout files, even those containing many shapes.

Evaluation

Implementation Details

The project was implemented using Python 3.9. Python was chosen due to its rich ecosystem for text processing and the availability of the standard library's turtle module for 2D visualization.

Dependency Management:

The only external dependency is Lark, which simplifies installation and ensures portability.

Interactive Mode:

An additional module, interactive_app.py, was developed using **Tkinter**. This integrates the compiler directly into a GUI, enabling a "Live Preview" mode similar to modern web development tools. Users can type code on the left panel and see the rendered output immediately on the right panel.

Testing Strategy

Testing was divided into two major categories: Smoke Tests and Functional Test Cases.

Smoke Tests

The compiler.py file contains an execution block that runs automatically when no arguments are provided. This serves as a lightweight continuous integration mechanism, verifying that core components (such as RectangleNode and Compiler) can be instantiated without runtime errors.

Functional Test Cases

The following table summarizes the test cases used to evaluate the correctness of the compiler:

Test ID	Category	Input Description	Expected Outcome	Result
TC-01	Basic Success	Rectangle with position, size, fill, and border	Draws a rectangle at specified coordinates	Pass
TC-02	Geometry	Circle with width and height defined	Draws an oval or circle centered correctly	Pass
TC-03	Core Shape	Line shape using ΔW and ΔH for end-point calculation	Draws a line segment correctly	Pass
TC-04	Syntax Error	Missing semicolon (rect...)	lark.UnexpectedToken or Syntax Error raised	Pass
TC-05	Attributes	Floating-point inputs (e.g., 10.5px) for size/position	Parsed correctly as float via the NUMBER terminal	Pass
TC-06	Edge Case	Width or Height = 0	AST creation succeeds, but the shape draws nothing	Pass
TC-07	Optional	Omitting outside color	Defaults to ast_nodes.Color('#000000') (Black)	Pass
TC-08	Semantic Error	Invalid color code (e.g., fill:#GGTTR)	LexerError or ValueError raised due to invalid regex/AST validation	Pass

Output Validation

A compilation sample from test_output.py confirms that the code generator correctly translates AST node attributes into Turtle commands, including the generation of rounded rectangles and proper color assignments.

Discussions

Design Justification

Use of Lark and a Custom AST:

Lark was selected because its LALR parser supports EBNF grammar and allows fast and reliable development of the language syntax. After parsing, the results are converted into custom AST nodes, which centralize all semantic validation—such as checking required attributes or verifying color formats. This clean separation ensures the parser only handles structure, while the AST handles rules and meaning.

Hand-Rolled Lexer:

Even though Lark provides its own lexer, the project includes a custom lexer in lexer.py. This gives full control over how tokens are generated and tracked, which is especially important for the verbose debugging mode (-v). The custom lexer provides a detailed token stream and accurate character positions, something that the default Lark lexer abstracts away.

Technical Challenges

Coordinate Transformation:

SimpleUI treats the top-left corner of the screen as the coordinate origin. Turtle graphics, however, use the center of the screen as the origin. This mismatch required implementing a coordinate conversion function inside the code generator so that shapes appear in the correct position when drawn.

Error Reporting:

The modules parser.py and compiler.py catch both Lark parsing errors and AST validation errors. To avoid overwhelming the user with Python stack traces, these low-level

exceptions are wrapped in a custom CompilerError. This provides clean, readable error messages that point directly to the problem in the SimpleUI code.

Current Limitations

The current version of SimpleUI includes several constraints that limit the complexity of layouts:

- No nesting:
  Shapes cannot be placed inside other shapes. Every element exists at the top level of the canvas.

- No variables or constants:
  Values such as sizes or colors must be repeated manually. There is no way to define reusable values within the .sui file.

Conclusion and Recommendation

Conclusion

The SimpleUI project successfully demonstrates the construction of a complete compiler pipeline. By combining Lark for parsing and Turtle for visualization, the system meets all functional requirements:

- Parses a custom DSL syntax
- Validates semantic rules through the AST
- Generates accurate rendered output
- Provides both CLI and GUI interfaces

The inclusion of an IDE-like interactive mode significantly enhances usability, showing that even a minimal DSL can serve as a practical layout language.

Recommendations

For future versions of SimpleUI, the following improvements are recommended:

1. Control Flow Support
   Add looping structures (e.g., repeat 5 { ... }) to simplify repetitive layout code.

   Variable Definitions
   Enable user-defined variables, such as:
   main_color: #FF0000;

2. This would improve readability and maintainability.

3. Rendering Optimization: Transition from Turtle—which is slow due to animated drawing—to a faster raster engine such as Pillow, PyGame, or Cairo.

Program Manual

Prerequisites

1. **Python:** Version 3.9 or higher.
2. **Lark:** Install the required dependency:

Command-Line Usage

Step 1: Save your SimpleUI code in a file, e.g., layout.sui.

Step 2: Open your terminal or command prompt.

Step 3: Run the compiler:

Execution: To run the generated script, which opens the Turtle canvas:
python my_output.py

Running the Interactive App (IDE Mode)

1. Launch: Start the live editor:
   python interactive_app.py
2. Usage: Click "Compile & Render" to execute the text and draw the output instantly on the canvas. The app includes an example layout ("WELCOME TO SimpleUI") that can be loaded to demonstrate functionality.

Appendices

Demo video:
https://drive.google.com/file/d/1Ja0n_qSa7_ZLCPtOMVGD_YtmpKaC1Re4/view?usp=sharing
Github: https://github.com/JustinHadinataCS/SimpleUI_CompTech