SVG Drawing Language Documentation

> This project was made partially with the help of AI.

1. General Explanation

This program is an interpreter for a simple, domain-specific language designed to generate Scalable Vector Graphics (SVG) files. It reads a source file written in this custom language, parses it, and executes the commands to produce a complete SVG image file as its output.

The process works as follows:

1. **Lexical Analysis (Lexing):** The `lexer.l` file defines rules to break the input text into a stream of tokens (e.g., keywords like `RECT`, numbers, variable names, operators).
2. **Syntactic Analysis (Parsing):** The `parser.y` file defines the grammar of the language. It takes the stream of tokens from the lexer and organizes them into a hierarchical structure called an **Abstract Syntax Tree (AST)**.
3. **Execution (Interpretation):** After the entire source file is successfully parsed into an AST, the program walks this tree. During the walk, it:
    - Manages variable declarations and assignments using a scoped symbol table.
    - Evaluates arithmetic expressions.
    - Executes control flow statements like `if` and `while`.
    - Translates drawing commands (`RECT`, `LINE`) into the corresponding SVG tag strings, which are printed to standard output.

The final output is a valid `.svg` file that can be rendered by any modern web browser.

2. Language Grammar

The language supports variable declarations, assignments, conditional logic, loops, and basic shape-drawing commands. The grammar can be summarized as follows:

```
Program ::= [CanvasDeclaration] StatementList

CanvasDeclaration ::= CANVAS Expression Expression

StatementList ::= Statement StatementList | <empty>

Statement ::=
    | VariableDeclaration
    | Assignment
    | DrawingCommand
    | IfStatement
    | WhileLoop
```

```
VariableDeclaration ::=
    | num ID = Expression
    | color ID = ColorExpression

Assignment ::= ID = Expression

DrawingCommand ::=
    | RECT Expression Expression Expression Expression [FillOption]
    | LINE Expression Expression Expression Expression [FillOption]

FillOption ::= fill = ColorExpression

IfStatement ::= if (Condition) { StatementList } [ else { StatementList } ]

WhileLoop ::= while (Condition) { StatementList }

Condition ::= Expression ComparisonOperator Expression

ComparisonOperator ::= > | < | == | != | >= | <=

Expression ::=
    | Term
    | Expression + Term
    | Expression - Term

Term ::=
    | Factor
    | Term * Factor
    | Term / Factor

Factor ::=
    | NUM
    | ID
    | ( Expression )

ColorExpression ::=
    | COLOR
    | ID
```

3. Input File Format & Language Features

An input file consists of a series of statements.

Comments

Single-line comments start with `//`.

```
// This is a comment and will be ignored.
```

Canvas

You can optionally set the canvas size. If omitted, it defaults to 21.0cm x 29.7cm (A4). This must be the first statement in the file.

```
// Sets the canvas to 50cm wide and 40cm tall
CANVAS 50 40
```

Data Types and Variables

There are two data types:

- `num`: A floating-point number.
- `color`: A string representing an SVG color. This can be a named color (e.g., `red`) or a hex code (e.g., `#FF0000`).

Variables must be declared before use.

```
// Declare a number variable
num counter = 0;
num width = 10.5;

// Declare a color variable
color background = "#EEE";
color stroke_color = "blue";
```

Assignment

Update the value of an existing variable.

```
num x = 5;
x = x + 10; // x is now 15
```

Expressions

The language supports standard arithmetic expressions with `+`, `-`, `*`, `/`. Parentheses `()` can be used to control the order of operations.

```
num x = (5 + 3) * 2; // x is 16
```

Drawing Commands

- `RECT x y width height [fill=color]` Draws a rectangle. The `fill` option is optional and defaults to black.
- `LINE x1 y1 x2 y2 [fill=color]` Draws a line. The `fill` option sets the line's stroke color and is optional (defaults to black).

```
RECT 1 1 5 3 fill=green;
LINE 0 0 10 10 fill=#FF0000;
```

Control Flow

Standard `if/else` and `while` loops are supported. They create new variable scopes.

- **If/Else Statement**

```
num x = 10;
if (x > 5) {
  RECT 0 0 5 5 fill=green;
} else {
  RECT 0 0 5 5 fill=red;
}
```

- **While Loop**

```
// Draw 5 rectangles in a row
num i = 0;
while (i < 5) {
  RECT (i * 2) 0 1.5 10 fill=blue;
  i = i + 1;
}
```

Example Input File (`drawing.txt`)

```
// A simple drawing with a loop and variables
CANVAS 30 30
```

```
color bg = #FAFAFA
color line_color = #333

// Draw a background rectangle
RECT 0 0 30 30 fill=bg

// Draw a series of shrinking, concentric squares
num i = 0
num size = 28
num offset = 1

while (i < 14) {
  RECT offset offset size size

  // Update variables for the next iteration
  size = size - 2
  offset = offset + 1
  i = i + 1
}

// Draw a red line across the middle
LINE 0 15 30 15 fill=red
```

there is block scoping

```
while (i < 5) {
    num x = x + i
}
num y = i // will result in error
```

4. How to Compile and Run

Prerequisites

You need `flex`, `bison`, and `gcc`. On a Debian/Ubuntu system, you can install them with:

```
sudo apt-get install flex bison build-essential
```

Required Files

Place the following files in the same directory:

1. `lexer.l` (The provided lexer definition)

2. `parser.y` (The provided parser definition)
3. `ast.h` (The header file defining the AST structures)
4. `Makefile` (not strictly necessary)
5. An input file (e.g., `drawing.svgl` from the example above)

Compilation

Run the following command in your terminal

```
make
```

Execution

Run the compiled program, feeding it your input file via standard input (`<`) and redirecting its standard output (`>`) to a `.svg` file.

```
./compiler < drawing.txt > output.svg
```

You can now open `output.svg` in any modern web browser (like Chrome, Firefox, or Safari) to see your drawing.