CFG for Minimal TypeScript

CS F365 – Compiler Construction

Phase 1: Language Design and Lexical Analysis

Grammar Notation

- *Uppercase* = Non-terminals
- `monospace` = Terminals (tokens)
- $|$ = alternatives, ε = empty production

Start Symbol & Program Structure

$$\begin{aligned}
\mathit{Program} &\rightarrow \mathit{StmtList} \\
\mathit{StmtList} &\rightarrow \mathit{Stmt}\ \mathit{StmtList} \mid \varepsilon \\
\mathit{Stmt} &\rightarrow \mathit{VarDecl} \\
&\quad \mid \mathit{AssignStmt} \\
&\quad \mid \mathit{IfStmt} \\
&\quad \mid \mathit{WhileStmt} \\
&\quad \mid \mathit{Block} \\
\mathit{Block} &\rightarrow \{\ \mathit{StmtList}\ \}
\end{aligned}$$

Variable Declaration

$$\begin{aligned}
\mathit{VarDecl} &\rightarrow \mathit{VarKeyword}\ \texttt{id}\ \texttt{:}\ \mathit{Type}\ \texttt{=}\ \mathit{Expr}\ \texttt{;} \\
&\quad \mid \mathit{VarKeyword}\ \texttt{id}\ \texttt{:}\ \mathit{Type}\ \texttt{;} \\
\mathit{VarKeyword} &\rightarrow \texttt{let} \mid \texttt{const} \\
\mathit{Type} &\rightarrow \texttt{number} \mid \texttt{string} \mid \texttt{boolean}
\end{aligned}$$

Assignment Statement

$$\mathit{AssignStmt} \rightarrow \texttt{id}\ \texttt{=}\ \mathit{Expr}\ \texttt{;}$$

Conditional Statement (if-else)

$$\begin{aligned}
\mathit{IfStmt} &\rightarrow \texttt{if}\ \texttt{(}\ \mathit{Expr}\ \texttt{)}\ \mathit{Block} \\
&\quad \mid \texttt{if}\ \texttt{(}\ \mathit{Expr}\ \texttt{)}\ \mathit{Block}\ \texttt{else}\ \mathit{Block} \\
&\quad \mid \texttt{if}\ \texttt{(}\ \mathit{Expr}\ \texttt{)}\ \mathit{Block}\ \texttt{else}\ \mathit{IfStmt}
\end{aligned}$$

Loop Statement (while)

$$\mathit{WhileStmt} \rightarrow \texttt{while (} \mathit{Expr} \texttt{) } \mathit{Block}$$

Expressions (with Precedence)

Precedence increases from top to bottom (lowest → highest).

$$\begin{array}{rcll}
\mathit{Expr} & \rightarrow & \mathit{Expr}\ \texttt{||}\ \mathit{AndExpr} & \leftarrow \text{Logical OR (lowest} \\
 & | & \mathit{AndExpr} & \\
\mathit{AndExpr} & \rightarrow & \mathit{AndExpr}\ \texttt{\&\&}\ \mathit{RelExpr} & \leftarrow \text{Logical AND} \\
 & | & \mathit{RelExpr} & \\
\mathit{RelExpr} & \rightarrow & \mathit{RelExpr}\ \texttt{relop}\ \mathit{AddExpr} & \leftarrow \text{Relational (==, !=,} \\
 & | & \mathit{AddExpr} & \\
\mathit{AddExpr} & \rightarrow & \mathit{AddExpr}\ \texttt{+}\ \mathit{MulExpr} & \leftarrow \text{Addition} \\
 & | & \mathit{AddExpr}\ \texttt{-}\ \mathit{MulExpr} & \leftarrow \text{Subtraction} \\
 & | & \mathit{MulExpr} & \\
\mathit{MulExpr} & \rightarrow & \mathit{MulExpr}\ \texttt{*}\ \mathit{UnaryExpr} & \leftarrow \text{Multiplication} \\
 & | & \mathit{MulExpr}\ \texttt{/}\ \mathit{UnaryExpr} & \leftarrow \text{Division} \\
 & | & \mathit{MulExpr}\ \texttt{\%}\ \mathit{UnaryExpr} & \leftarrow \text{Modulo} \\
 & | & \mathit{UnaryExpr} & \\
\mathit{UnaryExpr} & \rightarrow & \texttt{!}\ \mathit{UnaryExpr} & \leftarrow \text{Logical NOT} \\
 & | & \texttt{-}\ \mathit{UnaryExpr} & \leftarrow \text{Unary minus} \\
 & | & \mathit{Primary} & \\
\mathit{Primary} & \rightarrow & \texttt{id} \mid \texttt{number\_lit} \mid \texttt{string\_lit} \mid \texttt{true} \mid \texttt{false} \mid \texttt{(}\ \mathit{Expr}\ \texttt{)} &
\end{array}$$

Terminal Symbols Summary

Category	Terminals
Keywords	`let`, `const`, `if`, `else`, `while`, `true`, `false`
Types	`number`, `string`, `boolean`
Operators	`+`, `-`, `*`, `/`, `%`, `&&`, `||`, `!`
Relational	`==`, `!=`, `<`, `>`, `<=`, `>=`
Delimiters	`(`, `)`, `{`, `}`, `:`, `=`, `;`
Literals	`number_lit`, `string_lit`
Identifier	`id`

Lex Priority Order

Lex resolves conflicts by two rules — **longest match first**, then **order of appearance**. The recommended rule ordering is:

1. Multi-character operators (`==`, `!=`, `<=`, `>=`, `&&`, `||`)
2. Keywords (`let`, `const`, `if`, `else`, `while`, `true`, `false`)
3. Type keywords (`number`, `string`, `boolean`)
4. Literals (`number_lit`, `string_lit`)
5. Identifiers (`id`)
6. Single-character operators (`+`, `-`, `*`, `/`, `%`, `!`, `<`, `>`, `=`)
7. Delimiters (`(`, `)`, `{`, `}`, `:`, `;`)
8. Whitespace & Comments (skip)

How to Run the Lexer

Prerequisites

Ensure the following tools are installed on your system:

- `flex` — the lexer generator (implements Lex)
- `gcc` — the GNU C compiler

On Ubuntu/Debian, install them with:

```
sudo apt-get install flex gcc
```

Step 1: Generate C Code from the Lexer

Run `flex` on the `.l` file to produce `lex.yy.c`:

```
flex typescript_lexer.l
```

Step 2: Compile the Generated C Code

Compile `lex.yy.c` with `gcc` and link the flex library (`-lfl`):

```
gcc lex.yy.c -o ts_lexer -lfl
```

This produces the executable `ts_lexer`.

Step 3: Run on Test Files

Valid input program:

```
./ts_lexer valid_input.ts
```

Input program with lexical errors:

```
./ts_lexer error_input.ts
```

Alternatively, pipe input from stdin:

```
./ts_lexer < valid_input.ts
```

Expected Output Format

The lexer prints a formatted token stream to stdout:

```
=============================================================
        Minimal TypeScript Lexer  --  Token Stream
=============================================================
LINE   TOKEN TYPE            LEXEME
-------------------------------------------------------------
4      KW_LET                let
4      IDENTIFIER            x
4      COLON                 :
4      TYPE_NUMBER           number
4      OP_ASSIGN             =
4      NUM_LIT               10
4      SEMICOLON             ;
...
=============================================================
  Total tokens  : 87
  Lexical errors: 0
=============================================================
```

For the error input, unrecognized characters are reported inline:

```
7      *** LEXICAL ERROR ***       Unexpected character: ’@’
10     *** LEXICAL ERROR ***       Unexpected character: ’#’
...
  Lexical errors: 4
```

The lexer returns exit code `0` on success and `1` if any lexical errors were found, making it suitable for use in automated build pipelines.