Kotlin Compiler in Haskell

Group Members

- **Mário Minhava** - up202206190
- **Rodrigo Mendes** - up202308366

Introduction

This project is a simplified Kotlin compiler implemented in Haskell. The primary goal is to showcase compiler construction techniques using functional programming. The compiler is divided into three main components:

- **Lexer**: Handles lexical analysis and converts input source code into tokens.
- **Parser**: Analyzes the tokens based on grammar rules and constructs an Abstract Syntax Tree (AST).
- **Main Program**: Manages the compilation process, integrating both the lexer and parser.

Project Structure

Files

- **Lexer.x**: Defines the lexical analysis using `Alex`, a tool for generating Haskell lexers. This file specifies regular expressions to identify different types of tokens. The lexer handles:
  - Whitespace and comments.
  - Reserved keywords (`val`, `var`, `if`, `else`, etc.).
  - Identifiers, operators, and literal values (integers, strings, etc. . .).
- **Parser.y**: Implements the parsing logic using `Happy`, a parser generator for Haskell. The parser defines:
  - Grammar rules that specify the valid structure of Kotlin code, including statements, expressions, and control structures.
  - The construction of an Abstract Syntax Tree (AST), which represents the hierarchical structure of the input code.
  - Error handling mechanisms to report syntax errors with expected tokens.
- **Main.hs**: The main Haskell module that ties the lexer and parser together. It:
  - Reads the input source code.
  - Uses the lexer to generate a list of tokens.
  - Passes the tokens to the parser to build the AST.
  - Outputs the resulting AST to a file and prints it for inspection.

- **ast.txt**: Contains the output of the AST after parsing a sample input. This file is used for debugging and verifying the correctness of the AST generation, and building a hierarchical structure of the code we provided in the terminal.
- **trabalho.cabal**: Configuration file for the Haskell Cabal build system. It manages project dependencies, compiler options, and build settings, ensuring that all necessary libraries and tools are included.

Setup and Compilation

Prerequisites

- `Haskell-GHC`.
- `Alex` for lexical analysis and `Happy` for parsing.
- `Cabal` to manage the dependencies

Building the Project

1. Configure your `file.cabal` to install all needed packages and dependencies
2. On your terminal type `cabal build` thus, building the project

Running the Compiler

The compiler reads Kotlin source code from standard input, processes it through the lexer and parser, and outputs the Abstract Syntax Tree (AST). The AST provides a structured representation of the input program, which can be used for further stages like semantic analysis or code generation.

How to run the project - On your terminal type:

```
cabal clean
cabal run
```

You don't need to do cabal clean everytime, but it's a good practice to do it, since it cleans leftover files from cabal build

Implementation Details

Lexer (`Lexer.x`)

- Uses regular expressions to define token patterns.
- Handles whitespace, comments, keywords, identifiers, and literals.
- Generates tokens such as `TokenVal`, `TokenVar`, `TokenIf`, and `TokenElse`.
- Provides a streamlined way to identify and categorize parts of the Kotlin source code.

Parser (`Parser.y`)

- Defines the grammar rules for a subset of the Kotlin language.

- Uses the tokens produced by the lexer to build the Abstract Syntax Tree (AST).
- Supports basic Kotlin constructs, including variable declarations, assignments, control flow statements, and expressions.
- Implements error handling to provide feedback on syntax issues during parsing.

Main Module (`Main.hs`)

- Acts as the entry point for the compiler.
- Imports the lexer and parser modules.
- Reads input code, applies the lexer to tokenize it, and then parses the tokens to generate the AST.
- Outputs the AST for debugging and verification purposes.

.hs files

`Lexer.hs` and `Parser.hs` files can be found here:
For the Lexer.hs:
`dist-newstyle/build/x86_64-linux/ghc-9.4.8/trabalho-0.1.0.0/build/Lexer.hs`
For the Parser.hs:
`dist-newstyle/build/x86_64-linux/ghc-9.4.8/trabalho-0.1.0.0/build/Parser.hs`.

Error Handling

- **Lexer**: Skips invalid characters and handles unrecognized tokens gracefully.
- **Parser**: Provides clear syntax error messages, including line numbers and expected tokens, to help with debugging.

What can't our project do?

We implemented a few more things than the ones that were asked by the professor for the project, but things like: For loops, arguments on main function or class declarations don't work.

Things that went wrong

As in every project not everything goes according to plan, and this project was no exception. In the beggining we had a few headaches configuring the .cabal file, and problems with the newly added `build-tool-depends`.

After we managed to configure the files, fixing Parser errors was a pain, since there was no concrete display of the exact error.

We defined a few things in the beggining just to make it work and down the line it was a problem, and we had to redo them.

Future Improvements

- Extend the parser to support a wider range of Kotlin features, such as functions, classes, and advanced control structures (if needed).
- Implement semantic analysis to check for type errors and other semantic issues in the input code.
- Add a code generation phase to produce executable code.
- Improve the error reporting mechanism to provide more context and suggestions for fixing syntax errors as it was quite difficult to understand/correct errors.

Conclusion

This project demonstrates the implementation of a simple Kotlin compiler using Haskell. It showcases the use of `Alex` for lexical analysis and `Happy` for parsing, integrating these components into a cohesive compilation pipeline. The current implementation handles a basic subset of the Kotlin language and lays the groundwork for further enhancements, such as semantic analysis and code generation.

This project comes with great value as it shows us a "behind the scenes" of what's happening when we casually write code.