Kotlin Compiler

This project was developed for the Compilers (CC3001) course at the Faculty of Sciences of the University of Porto (FCUP). It is a compiler for a previosly defined subset of the Kotlin programming language, explained in depth in the Coverage section.

The entire compiler is implemented in Haskell, using two key tools: **Alex** and **Happy**. Their roles and applications within the project are covered in detail in the Tools section.

Table of Contents

Features

- **Lexical Analysis**: The compiler's lexical analysis scans the Kotlin source code and converts it into a sequence of tokens, representing elements such as keywords, identifiers, literals, and operators. This step breaks down the code into manageable components for subsequent stages.

- **Parsing**: In the parsing phase, the compiler processes the tokens generated during lexical analysis to construct an abstract syntax tree, which reflects the hierarchical structure of the code according to the subset's grammar. This tree enables the compiler to understand the nested relationships among expressions, statements, and blocks.

- **Semantic Analysis**: Semantic analysis ensures the logical correctness of the syntax tree. This phase includes type checking, verification of variable declarations, and validation of proper usage for expressions and operators within the Kotlin subset. It catches errors that are beyond the reach of parsing alone.

- **Code Generation**: The code generation phase translates the validated syntax tree into an intermediate representation or target code, producing output that represents the original Kotlin logic within the defined subset and is suitable for execution.

- **Liveness Analysis**: Consists in creating a graph of intersections and living variables in moment of entry and leaving of each instruction of intermediate code in order to determinate the graph of intersections .

- **Register Allocation**: Determines the graph of intersections (non-directed graph) which each node is a register, then we put the node with less neighbours in stack and removes the node from graph and the incident archs until the graph is empty then pop from stack and color it different of their neighbours until stack is empty, each register would have an color.

- **Assembly Code Generetor**: Uses the Register Allocation and the Intermidiate Code to generate assembly code instructions in MIPS MARS assembly (uses some pseudo-instructions) generates the .data for float and strings and then .text (generates functions to print and read to)

Coverage

The compiler processes Kotlin code with the following features: arithmetic expressions, boolean expressions, the `print` and `readln` standard functions, variable declarations, assignments, conditional expressions (including `if` and `if-else` statements), and `while` loops.

We do not support function declarations, imports, or external libraries, nor do we handle object-oriented constructs such as classes or interfaces. Consequently, each file compiled with this tool must start with a `main` function declaration, which serves as the sole entry point for all executable code. This `main` function cannot accept parameters, and all code in the file must be contained within it.

Tools

- **Alex** is a lexer generator that processes source code into tokens, capturing individual language elements like keywords, identifiers, and symbols.

- **Happy** is a parser generator that constructs the syntax tree from the tokens Alex generated, allowing us to validate and interpret the structure of Kotlin code within the defined subset.

Together, Alex and Happy create a cohesive compilation pipeline that translates Kotlin code into an intermediate form, ready for further analysis and compilation.

Prerequisites

Ensure the following tools are installed before building and running the project:

1. **GHC v.9.4.8 (or above)**

   GHC (Glasgow Haskell Compiler) is required to compile and run Haskell code.

   - **Installation**: Visit the GHC download page (https://www.haskell.org/ghc/download.html) and select the installer for your operating system.
   - **Verification**: Run `ghc --version` in your terminal to confirm installation.

2. **Cabal v.3.10.3.0 (or above)**

Cabal is a package manager for Haskell projects, used to build and manage dependencies.

- **Installation**: Download Cabal from the official Cabal website (https://www.haskell.org/cabal/download.html) and follow the installation instructions for your OS.
- **Verification**: Run `cabal --version` to ensure Cabal is properly installed.

Installation

1. Clone the repository from GitHub:

```
git clone https://github.com/ricardofig016/kotlin-compiler
cd kotlin-compiler
```

2. Install the dependencies:

```
cabal update
cabal install
```

Usage

To compile Kotlin files with this project, follow these steps:

1. **Build the Project**

   Use the following command to build the project:

```
cabal build
```

2. **Prepare the Input File**

   Ensure your Kotlin file is named in the format `input[x].kt`, where `x` is an integer (e.g., `input1.kt`), and place it in the `inputs` directory.

3. **Run the Project**

   Run the project with the following command, passing the integer `x` to specify the file:

```
cabal run .\kotlin-compiler.cabal [x]
```

   For example, to compile `input1.kt`, use:

```
cabal run .\kotlin-compiler.cabal 1
```

The compiler will process the specified Kotlin file and generate `output.txt`, which includes the token sequence and abstract syntax tree (AST) representation.

Project Structure

```
kotlin-compiler
├── kotlin-compiler.cabal
├── output.txt
├── README.md
├── inputs
│   ├── input1.kt
│   ├── input2.kt
│   └── (...)
└── src
    ├── CodeGeneretor.hs
    ├── InstructionSelector.hs
    ├── Lexer.hs
    ├── Lexer.x
    ├── Liveness.hs
    ├── Main.hs
    ├── Parser.hs
    ├── Parser.y
    ├── SemanticAnalyzer.hs
    └── SymbolTable.hs
```