Assignment 3 : Kaleidoscope and MLIR

Puneet Mangla (CS17BTECH11029)

1. An **intermediate representation (IR)** is a code internally used by compilers to represent source code. It is then further used by compilers to produce machine dependent code after processing and optimizing it. An IR needs to be machine independent and accurate to be capable of representing source code without any error and loss of information. We need IR of a source code because we can apply any possible optimisations without caring about hardware since it is machine independent.

2. **LLVM-IR** is an intermediate code representation that resembles to low-level programming language like assembly and capable of representing 'all' high-level languages cleanly for efficient compiler transformations and analysis. The low level representation of LLVM-IR allows many source languages to be mapped to it, making it 'Universal-IR'. **Design Philosophy :** LLVM-IR is an Statically Single Assignment (SSA) based representation that provides type safety, low-level operations and flexibility. It is designed to be used in three different forms: as an in-memory compiler IR, as an on-disk bitcode representation (suitable for fast loading by a Just-In-Time compiler), and as a human readable assembly language representation. It aims to be light-weight and low-level while being expressive, typed, and extensible at the same time.

   MLIR is an intermediate representation similar to LLVM-IR, but introduces notions from polyhedral loop optimization as first-class concepts. It is also a collection of compiler utility tools that is between model representation and and low level programs that generate machine code. **Design Philosophy :** This is designed as a hybrid, able to represent, analyze and transform high level dataflow graph as well as target-specific code generated for high performance data parallel systems by combining multiple levels of abstraction into a single compilation unit. Such abstractions include TensorFlow operations, nested polyhedral loop regions, LLVM instructions. MLIR is designed to be used in three different forms: a human-readable textual form suitable for debugging, an in-memory form suitable for programmatic transformations and analysis, and a compact serialized form suitable for storage and transport.

3.

Lexer :

- Both the lexers use an enum of tokens with `tok_eof`, `tok_def`, `tok_extern`, `tok_identifier`, `tok_number` as common tokens. Toy include more tokens for `semicolons, parenthesis, return` etc. Moreover Toy Lexer is object oriented whereas Kaleidoscope Lexer isn't.
- Both uses `std::string IdentifierStr` and `double NumVal` for storing name of identifiers and numerical values.
- The working of `getTok()` function in Toy Lexer and `gettok()` in Kaleidoscope lexer have similar definitions. They first remove and leading whitespaces and then consume character by character to look for identifier and return the respective token. `ASCII` value is return for unknown tokens. If `EOF` is consumed both returns `tok_eof`.
- In Kaleidoscope, variables `IdentifierStr` and `NumVal` are global whereas Toy Lexer they are private members of a class.
- Toy Lexer has additional variables for storing line and column number (`int curLineNum, int curCol`) and functions (`getNextChar()`) for incrementing them. It also used `llvm::StringRef` also.

Parser :

- Both the parsers makes an abstract syntax tree and use top level approach of parsing. Also both are object oriented.
- Both of them uses derived classes of `ExprAST` to define classes corresponding to functions, variables, literals, return statements, function calls. The derived classes of `ExprAST` in Toy parser contains a variable `Location` to store the location of that particular piece of code and `getKind()` function to tell the type of the class. For examples .

```
class NumberExprAST : public ExprAST {
     double Val;
     Public:
          NumberExprAST(Location loc, double Val) :
     ExprAST(Expr_Num, loc), Val(Val) {}
     ...
}
```

- Both the parsing approaches use recursive descent parsing strategy. Every parser returns some derived class of `ExprAST`. For Example

```
static std::unique_ptr<ExprAST> ParseParenExpr() {
      getNextToken(); // eat (.
      auto V = ParseExpression();
```

```
        if (!V)
      return nullptr;
      if (CurTok != ')')
           return LogError("expected ')'");
      getNextToken(); // eat ).
      return V;
}
```

- The `ProtoTypeAST` class in Toy parsers uses a vector of class `VariableExprAST` to represent list of arguments whereas Kaleidoscope parser only uses a vector of strings to represent them.
- The `FunctionAST` class in Toy parser uses a class `ExprASTList` to represent the function body whereas Kaleidoscope parser only uses a class `ExprAST` for defining a function body.