Assignment 3: Kaleidoscope and MLIR

Sagar Jain
CS17BTECH11034

April 12, 2019

1. What is an IR? Why is it required?
   Ans: IR stands for Intermediate Representation. An intermediate representation is an alternate representation of a program between the source and target languages.

   Conversion of source to an IR before the target is required for the following reasons:

   - Conversion to IRs helps break down the problem of translation into smaller, easier steps. For Example, lexical analysis and parsing as seperate steps is much more feasible than doing both together.
   - Optimisations can be split into machine independent and machine dependent which would not be possible in case there were no IRs.
   - Using IRs helps make retargetable compilers, i.e. same backend can be used for multiple languages and/or same frontend can be used for multiple different machines.

2. What is LLVM-IR? Why is its main purpose? What is its design philosophy? What is MLIR? What is the main design philosophy of MLIR?
   Ans: LLVM IR is a low-level intermediate representation used by the LLVM compiler framework. The main purpose of LLVM IR is to represent all high level languages in LLVM assembly language. It is the common code representation used throughout all phases of the LLVM compilation strategy.

   The design philosophy of the LLVM IR *is to be light-weight and low-level while being expressive, typed, and extensible at the same time. It aims to be a a universal IR such that all high level ideas can be mapped to it cleanly.*

   MLIR stands for one of "Multi-Level Intermediate Representation". MLIR is a representation format and library of compiler utilities that sits between the model representation and low-level compilers/executors that generate hardware-specific code.
   MLIR is intended to be a hybrid IR which can support multiple different requirements in a unified infrastructure. The MLIR project aims to define a common intermediate representation (IR) that will unify the infrastructure required to execute high performance machine learning models in TensorFlow and similar ML frameworks.

3. Read the Lexer and Parser codes of Kaleidoscope and Toy (from MLIR). Post some notes on the same. The notes should show some familiarity between of the internals of Lexer and Parser of both the language.

Lexer

The lexer for Kaleidoscope and Toy are very similar, we expore the similarities in the following points.

(a) Both the languages have a precise set of possible tokens:
   i. Kaleidoscope has eof, def, extern, identifier, number and unknown characters.
   ii. Toy has eof, return, var, def, identifier, number, semicolons, square brackets, parantheses and brackets.

(b) Both the lexers have a function ***gettok*** which repeats the following steps to get tokens:
   i. Skips any whitespace.
   ii. Checks if next character is of type ***char***, if yes goes on reading all further input of letters, numbers and underscores appending them onto the string, once done it does the following:
      - Checks if the composed string belongs to the set of reserved words, if yes returns the token corresponding to the reserved word, for example, ***tok_def***.
      - If no reserved matches the string, then it is returned as an identifier token using for example, ***tok_identifier***.
   iii. If the next character is a number, the lexer keeps reading all following input as long as it is a digit or a decimal point and keeps appending it to a string, when done, this string is converted to a number using ***strtod*** and then returned as number token, i.e. ***tok_number***.
   iv. If the next character is a # then, the lexer keeps reading till EOF or newline is encountered, if EOF is encountered it returns the EOF token and if newline is encountered, then the next token is returned.
   v. Else, just the ascii value of the token is returned.

(c) Both the languages have essentially the same lexer, the same functions and the same methedology to split into tokens, the only thing extra in Toy is that it has explicit tokens for braces, parantheses and square brackets.

Parser

Both the languages also have very similar parsers, we note the important points in the following lines:

- Both Kaleidoscope and Toy have recursive descent parsers.
- Both the parsers use ***getNextToken*** to get the next token from the lexer . Kaleidoscope uses CurTok to store it while Toy uses ***getCurToken*** to access it.
- Both use ***getTokPrecedence*** to get the precendence of a binary operator. The precendence is exactly the same for all arithmetic operators. (* > + = -)
- Both the parsers use seperate functions to parse every production rule of the grammar.
- The production rules implemented using fucntions are also very similar in both the languages as seen below:

Rule	Toy	Kaleidoscope	Note
Number Litral	numberexpr ::= number	numberexpr ::= number	
Identifier	identifierexpr ::= identifier ::= identifier '(' expression* ')'	identifierexpr ::= identifier ::= identifier '(' expression* ')'	To consume the '(', Kaleidoscope simply uses getNextToken(), while Toy uses consume(Token('')'))
Primary Expression	primary ::= identifierexpr ::= numberexpr ::= parenexpr ::= tensorliteral	primary ::= identifierexpr ::= numberexpr ::= parenexpr	The only extra rule is tensor literal in Toy.
Expression	expression::= primary binoprhs	expression::= primary binoprhs	

- The production function and rules for binoprhs are also exactly the same.
- The way ***prototypes*** are defined may seem different but they both boil down to the same thing. Kaleidoscope:
  prototype ::= id '(' id ')'*
  Toy:
  prototype ::= def id '(' decl_list ')'
  decl_list ::= identifier or identifier, decl_list
- There are some differences in how ***defs*** are defined, variable declaration, toy having extra constructs like **shape_lists**, etc.
- Other than the added support for tensor/arrays, the expression parsing is also very similar.