Compilers-I

CS 3510 Spring 2019

Reading Assignment 3:

Kaleidoscope and MLIR

INSTRUCTOR: DR. Ramakrishna Upadrasta

By:
Vijay Tadikamalla
CS17BTECH11040

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

1. **What is IR? Why is it required?**
   IR (intermediate representation) is a data structure/code used internally by a compiler or virtual machine for representation of a program between the source and target languages
   IR is required because:

    1. It helps to break the difficult problem of translation into two simpler, more manageable pieces.
    2. It helps in building retargetable compilers. (A retargetable compiler is a compiler that has been designed to be relatively easy to modify to generate code for different CPU instruction set architectures). So using IR:
        - We can build new back ends for an existing front end.
        - We can build a new front-end for an existing back end (so a new machine can quickly get a set of compilers for different source languages).
        - For n source language and m target language, we will only have to write (n+m) half compilers instead of n*m full compilers. (Assumption: We are able to write a good IR that is fairly independent of the source and target languages)
    3. To perform machine independent optimizations.

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?**

   LLVM IR is a low-level intermediate representation used by the LLVM compiler framework. LLVM IR can be thought of as a platform-independent assembly language with an infinite number of function local registers.

   The main purpose of LLVM-IR is to connect the frontends and backends, allowing LLVM to parse multiple source languages and generate code to multiple targets. Frontends produce the IR, while backends consume it. The IR is also the point where the majority of LLVM target-independent optimizations takes place.

Some design philosophies:

1. The LLVM-IR aims to be light-weight and low-level while being expressive, typed, and extensible at the same time.
2. It aims to be a “universal IR” by being at a low enough level such that high-level ideas may be cleanly mapped to it.

MLIR is a common intermediate representation (IR) that will unify the infrastructure required to execute high-performance machine learning models in TensorFlow and similar ML frameworks. It is a common IR that also supports hardware-specific operations.
Its main design philosophy is:

1. To reduce the cost to bring up new hardware, and improve usability for existing TensorFlow users.
2. To be a hybrid IR which can support multiple different requirements in a unified infrastructure

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 the internals of Lexer and Parser of both the language.**

Kaleidoscope
Lexer

- The lexer returns tokens [0-255] if it is an unknown character, otherwise, it will either be one of the Token enum values

```
enum Token {
  tok_eof = -1,

  // commands
  tok_def = -2,
  tok_extern = -3,

  // primary
  tok_identifier = -4,
  tok_number = -5,
};

static std::string IdentifierStr; // Filled in if tok_identifier
static double NumVal;             // Filled in if tok_number
```

- The lexer is a single function named gettok().
- The gettok function is called to return the next token from standard input.
- It needs to recognize identifiers and specific keywords like “def” and “extern”.

Regex:
identifier: [a-zA-Z][a-zA-Z0-9]*
Number: [0-9.]+

Parser

- It has basic parsing mechanisms for:
    - Numeric literals - A routine ParseNumberExpr() is called when the current token is a tok_number token.
    - Handling variable references and function calls - A routine ParseIdentifierExpr() is called when the current token is a tok_identifier token. It has recursion and error handling
    - Arbitrary primary expression - We need a helper function to determine what sort of Arbitrary primary expression it is. So a helper function ParsePrimary() is used here.
    - Binary Expressions - Binary expressions are significantly harder to parse because they are often ambiguous. So the parser uses a technique called Operator-Precedence Parsing. This parsing technique uses the precedence of binary operators to guide recursion.
    - Handling of function prototypes - Kaleidoscope uses ‘extern’ for both function declarations as well as function body definitions. So a routine ParsePrototype() is used here.

MLIR
Lexer

- This is the list of Tokens returned by Lexer

```
enum Token: int {
  tok_semicolon = ';',
  tok_parenthese_open = '(',
  tok_parenthese_close = ')',
  tok_bracket_open = '{',
  tok_bracket_close = '}',
  tok_sbracket_open = '[',
  tok_sbracket_close = ']',

  tok_eof = -1,

  // commands
  tok_return = -2,
  tok_var = -3,
  tok_def = -4,

  // primary
  tok_identifier = -5,
  tok_number = -6,
};
```

- Here getTok() function returns the next Token from the standard input.
- It needs to identify keywords like “return”, ”def”, ”var”.
- **Regex**
    - Identifier: [a-zA-Z][a-zA-Z0-9_]*
    - Number: [0-9.]+

<u>Parser</u>

Apart from the parsing mechanisms mentioned above, the toy language uses some more parsing mechanisms. Some of them are mentioned below.

- A Parser for a full module. (A module is a list of function definitions)
- A Parser for a return statement
- A Parse a literal array expression.

Similarity

Lexer

- A similar strategy is used for mapping tokens to numbers.
- Both have almost Regular expressions for identifiers and numbers. The only exception is that toy language allows underscores(_) in identifier naming.
- The function getTok() is used to obtain the next token from the standard input.
- The similar approach/implementation is used for:
  - Skipping whitespaces
  - Identifying keywords
  - Comments
  - EOF

Parser

- Both make different parsing functions for parsing properties
- Both share almost same parsing mechanism for parsing Numerical literals, handling function calls, binary expressions, etc.
- Both use operator precedence while parsing binary operator.

References:

http://cs.lmu.edu/~ray/notes/ir/
https://blog.gopheracademy.com/advent-2018/llvm-ir-and-go/
https://hub.packtpub.com/introducing-llvm-intermediate-representation/
https://llvm.org/docs/LangRef.html#abstract
https://github.com/tensorflow/mlir
https://llvm.org/docs/tutorial/LangImpl01.html
https://github.com/tensorflow/mlir
https://llvm.org/docs/tutorial/LangImpl01.html
https://github.com/tensorflow/mlir/blob/master/g3doc/LangRef.md
https://github.com/tensorflow/mlir/blob/master/g3doc/Tutorials/Toy/Ch-1.md
https://github.com/tensorflow/mlir/blob/master/examples/toy/Ch1/include/toy/Lexer.h
https://github.com/tensorflow/mlir/blob/master/examples/toy/Ch1/include/toy/Parser.h