Abstract:

Protocol buffers (Protobuf) is a novel way of serializing structured data, which can be used in communication protocols or data storage. In this project, I implement a programming language, PF, to manipulate Protobuf data directly. It treats Protobuf objects as its fundamental elements and provides built-in operators and functions to operate on these objects. By using PF, we do not need to worry about the different access functions to handle different fields of a Protobuf data. PF is an new language aiming to facilitate the data operations with Protobuf and to provide a intuitive idea of how to use it.

Introduction:

Protocol buffers is powerful and useful in developing programs for communicating and storing structured data, given the trends of big data. It is language-neutral and platform-neutral. Google developed Protobuf with the goal of providing a better way to communicate between systems. It shares some commons with JSON or XML but it is more efficient and flexible.

To use Protobuf, we can add a message contains several name-value pairs in the .proto file. The message describes the data structure we want to serialize, which is an aggregate containing a set of typed fields. The types could be bool, int32, float, double, string and even message types. Fig.1 is an example of a .proto file. After defining the how our data is structured in Protobuf, we easily write and read to and from our structured data using a variety of languages, such as C++, Java, JavaScript, Ruby, Python and Go. We can use Protocol Compiler to generate corresponding classes or objects and the generated classes or objects provide getters and setters to deal with protobuf data with an efficient binary encoding format. Importantly, we can still read data encoded with old format after extending its format.

package tutorial;

message Person {

required string name = 1;

required int32 id = 2;

optional string email = 3;

enum PhoneType {

MOBILE = 0;

HOME = 1;

WORK = 2;

}

message PhoneNumber {

required string number = 1;

optional PhoneType type = 2 [default = HOME];

}

repeated PhoneNumber phones = 4;

}

message AddressBook {

repeated Person people = 1;

}

While these languages have libraries to handle Protobuf data, they are not native. This means that Protobuf is not something fundamental to the language, like the function to the functional language. Moreover, sometimes we may find that accessing elements of a Protobuf data is difficult especially when it contains message types since the access functions could be different for different field types and for repeated fields. PF, a language I implemented in this project, is oriented to handle and manipulate Protobuf data directly. It is designed to be simple, intuitive and specific, and to help developers quickly master how to handle Protobuf data. It has some primitive data types, number, string, bool and a complex data type to store protobuf data object. It provides some built-in functions to operate these data types.

Approach:

The project was developed on a combination of Mac OS X and Ubuntu. The complier was written in OCaml and was developed using vim. The PF code is compiled to C++, allowing for an high-performance execution, and then g++ was used to compile the code to an executable file. Makefiles and Bash scripts were used to compile and test these codes automatically. The architecture on the PF language compiler is shown in Fig.2.

1. Lexer

The lexer was implemented using Ocamllex and the corresponding file is lex.mll. The lexer will take a PF program as input and generate token streams according to the defined lexical rules as output. It will reject programs with illegal syntax. It will also discard unnecessary information such as white space and comments.

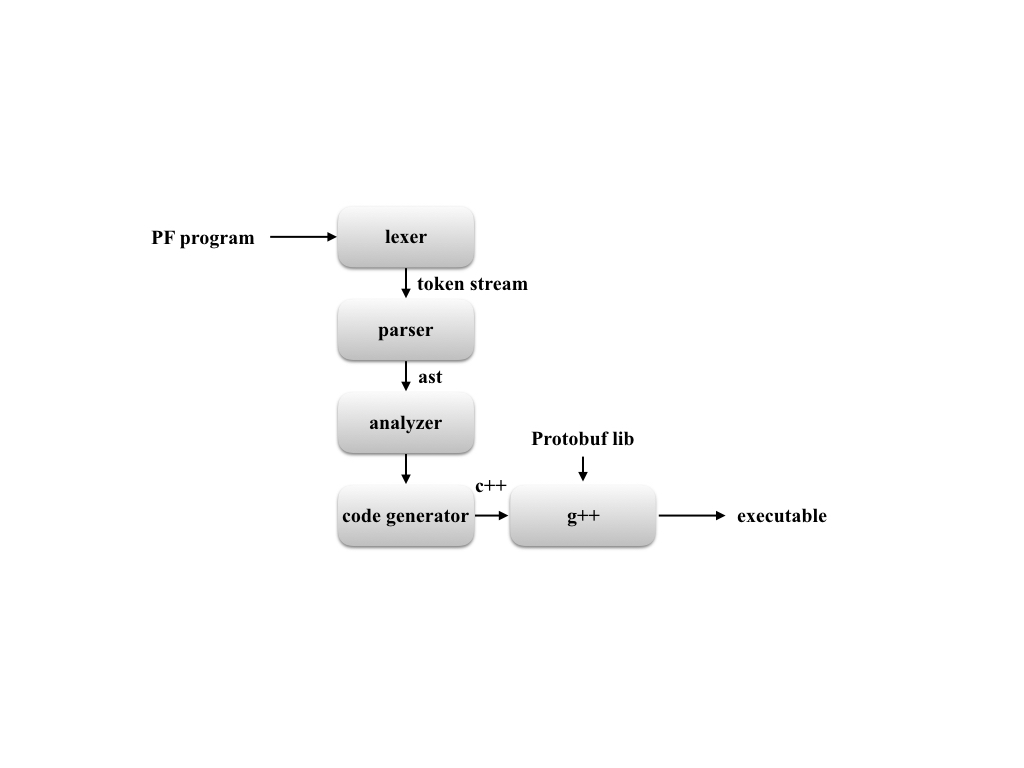
2. Parser

The parsers was implemented using ocamlyacc and the corresponding file is parse.mly. It will take the token streams provided by the lexer as input and generate an abstract syntax tree(AST) as output. The AST was parsed using the defined grammatical rules and this process will reject programs which cannot meet he rules.

3. Analyzer

The analyzer was implemented in Ocaml and the corresponding file is analyzer.ml. The analyzer take the AST provided by the parser as input and check the semantic requirement. The semantic checking includes checking on :

1) type requirements:

e.g. We can only assign an expression to a variable which have been declared to the same type;

Arithmetic and comparison operators work on number types only;

Logical operators work on boolean types only;

Protobuf element access operator works on Protobuf variables only.

2) declaration requirements:

e.g. We can only use a variable that has been declared;

We cannot declare a variable that has been declared before.

3) source file requirement:

e.g. The program should provide valid .proto files to declare Protobuf variavles.

4) scope requirements:

e.g. We cannot use a variable that is declared in another scope but has not been declared in its own scope.

5) language-specific requirements:

e.g. The Protobuf elements access statement should be valid.

4. Generator

The generator was implemented in Ocaml and the corresponding file is pf.ml. After checking the semantic requirements by analyzer.ml, pf.ml take the AST as input and generate C++ code as its output.