**2. Language Tutorial**

**2.1 Environment Setup**

The compiler has been built a tested using an Ubuntu 16.04 virtual machine. All the testing and

the coding were done on this VM. Opam was first installed on the virtual machine and then

Ocaml, and Llvm are installed using opam package. The instruction on how to install the opam and LLVM under Ubuntu 16.04 is shown below:

sudo apt-get install -y ocaml m4 llvm opam

opam init

opam install llvm.3.6 ocamlfind

eval `opam config env`

Liva could also be run in OS X. For installation under OS X, please refer to the MicroC compiler’s README file which can be found at <http://www.cs.columbia.edu/~sedwards/classes/2016/4115-summer/index.html>

**2.2 Run and test**

To run and test in Ubuntu 16.04 virtual machine, you need change directory to folder 'Liva':

To test the compiler use test suit, follow the instructions below:

$ make liva

-ocamlfind ocamlopt -c -package llvm ast.ml

-ocamlfind ocamlopt -c -package llvm sast.ml

-ocamlyacc parser.mly

-ocamlc -c ast.ml

-ocamlc -c parser.mli

-ocamlfind ocamlopt -c -package llvm parser.ml

-ocamllex scanner.mll

127 states, 6605 transitions, table size 27182 bytes

-ocamlfind ocamlopt -c -package llvm scanner.ml

-ocamlfind ocamlopt -c -package llvm semant.ml

-ocamlfind ocamlopt -c -package llvm codegen.ml

-ocamlfind ocamlopt -c -package llvm liva.ml

-ocamlfind ocamlopt -linkpkg -package llvm -package llvm.analysis ast.cmx sast.cmx parser.cmx -scanner.cmx semant.cmx codegen.cmx liva.cmx -o liva

You may need run chmod +x testall.sh to unlock the shell script.

$ ./testall.sh

-n test-add...

OK

-n test-and...

OK

...

-n fail-sub...

OK

-n fail-while1...

OK

To write a small program and test it:

test.liva:

class calculator{

int addition(int x, int y){

int z;

z = x + y;

return(z);

}

}

class test {

void main(){

int result;

class calculator obj = new calculator();

result = obj.addition(31,79);

print ("result=",result,"\n");

}

}

$ ./liva < test.liva > test.ll

$ lli test.ll

result=110

**2.3 Hello World**

To test Hello World, you need to first use make in the Liva directory. This will

compile all the necessary components to translate Liva files to LLVM IR.

Example: ‘test-hello.liva’

class test {

void main () {

print ("Hello World!");

}

}

There are two steps to running this program, assuming test-hello. liva is located in the

Liva directory:

1) Run ./liva <test-hello.liva

2) Run lli test-hello.ll

This should print out the desired result “Hello World!”.

**2.3 Basics**

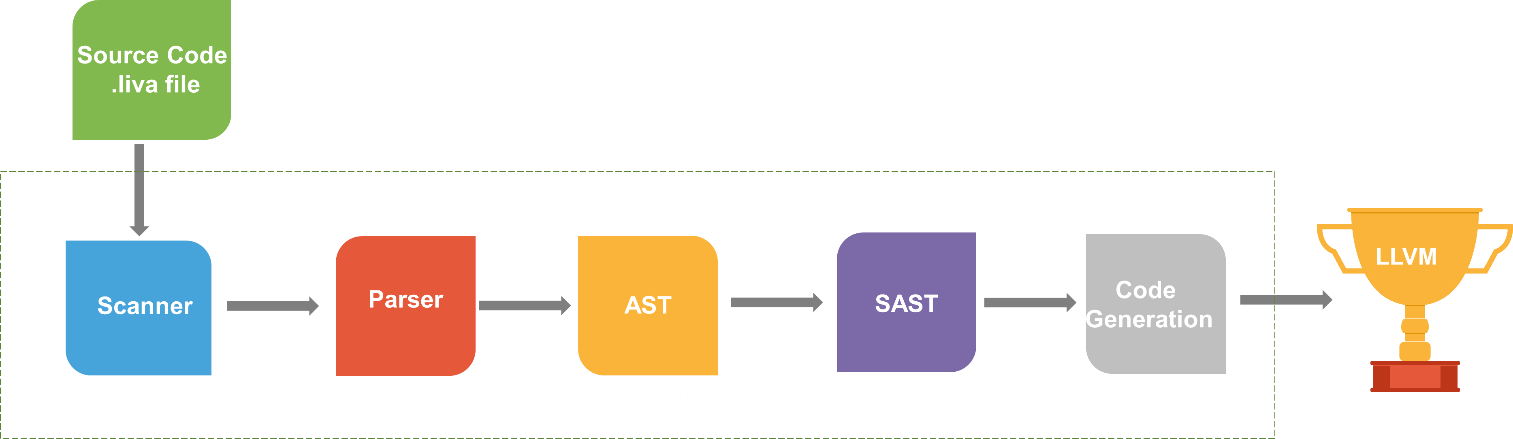
**2.4 Control flow**

**2.5 Defining methods**

**2.6 Classes and Inheritance**

**5. Architecture Design**

**5.1 Overview**



**Figure 5.1 Overview of compiler architecture design**

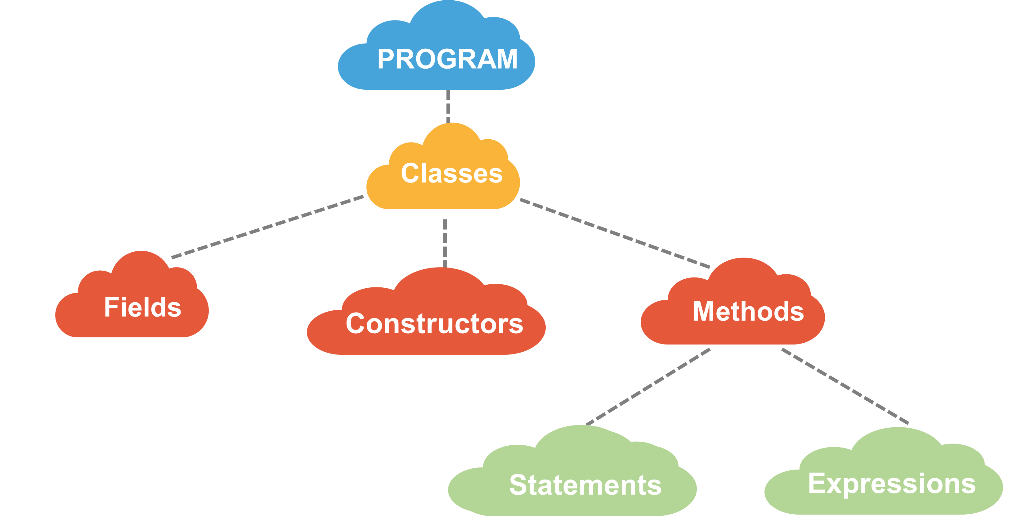
The compiler architectural design of Liva is shown in Figure 5.1. Overall, Liva follows a traditional compiler architecture design with a lexical scanner and parser at the front end, followed by generation of a Semantically Checked and Typed Abstract Syntax Tree (SAST) from an abstract syntax tree (AST) and finally LLVM IR code generation. We have a total of 5 modules which are codegen.ml, liva.ml, semant.ml, parser.mly, scanner.mll and 2 interfaces which are ast.ml and sast.ml.

**5.2 The scanner**

As the start of the front end, scanner reads a source file and convert it into tokens, and at the same time it checks whether each tokens is valid, if not, it will report the illegal character. Besides, it is also responsible for ignoring white space and comments which are not useful for Liva program.

**5.3 The Parser**

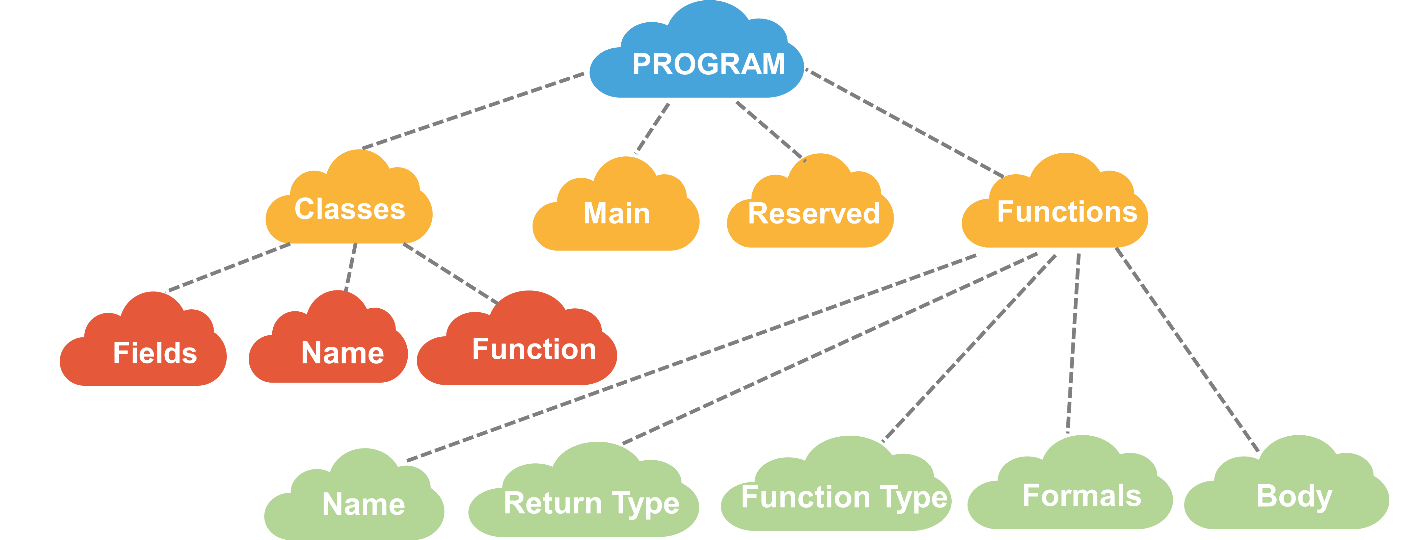
the tokens passed by the scanner are interpreted by the parser according to the precedence rules of Liva language and constructs an abstract syntax tree based on the definitions provided and the input tokens are constructed. The parser’s main goal is to organize the tokens of the program into class declarations. The top level of the abstract syntax tree is a structure called Program which contains all classes. The fields, constructors and methods are declared within the classes. Specific to the method declarations record is the creation of an AST of functions from groups of statements, statements evaluating the results of expressions, and expressions formed from operations and assignments of variables, references and constants. The Parser produces the abstract syntax tree (AST) which is shown in Figure 5.1.

****

**Figure 5.2 AST program representation.**

**5.4 The Semantic Checker**

There are four kinds of work a semantic checker is responsible for. To begin with, it adds reserved functions as a part of the SAST shown in figure5.3, and these reserved functions can also help to check whether there are any functions use the same name as reserved ones which is not allowed in Liva. Next, semantic checker do some work concerning statistic semantic checking. It checks whether the source code is semantically correct from various aspects including whether there are duplicated fields or methods in one class, whether there are duplicated classes in one program and adds default constructor if there isn’t user-defined constructor in one class. Thirdly, on the basis of the first two semantic checker deals with inheritance. Semantic checker finds all the inheritance relationship by looking through all the classes, if there is a inheritance relationship between two classes, semantic checker will add the fields and the methods of super class to subclass, but it the subclass declares fields or methods which share the same name as those of super class, subclass will override those fields and methods of super class. Finally, semantic checker converts its input, AST, to SAST which is helpful for code generator to generate LLVM IR code. Semantic checker separates the methods from classes, separates main method which is the entrance of Liva program from methods and add types to all the statements and expressions, in the meanwhile, it also do some work related with statistic semantic checking, including whether names or identifiers are defined before they are referred to, whether names or identifiers are used correctly, whether types are consistent and so on to make it as smooth as possible for code generator to generate code.

****

**Figure 5.3 SAST representation.**

**5.5 The Code Generator**

The main function of the compiler is to convert the abstract syntax tree into LLVM IR. After the semantic abstract syntax tree is generated, the semantic checker sends it to the code generator which construct the LLVM IR file which contains the final instructions for the program.

This code generation is written using the OCaml LLVM library, which uses OCaml functions to produce the desired LLVM code with the static variables used during code generation. Codegen.the\_module is the top-level structure that the LLVM IR uses to contain code and it contains all of the functions and global variables in a chunk of code. The Codegen.builder object is a object that keeps track of the current place to insert instructions and has methods to create new instructions. The Codegen.named\_values map keeps track of which values are defined in the current scope and what their LLVM representation is. After all of above is setup, the code generator iterates through the entire semantic abstract syntax tree and produces the necessary LLVM code for each function, statement, and expression.