Implementing a VSOP Compiler

Henry Leclipteur

May 2024

Lexical Analysis

I used the given flex/bison example to implement the lexical analysis part.
Most of my work is in the file lexer.lex.
This part was about catching sequence of characters and save them as tokens, the 2 main difficulties were the multiple lines comments and the strings.

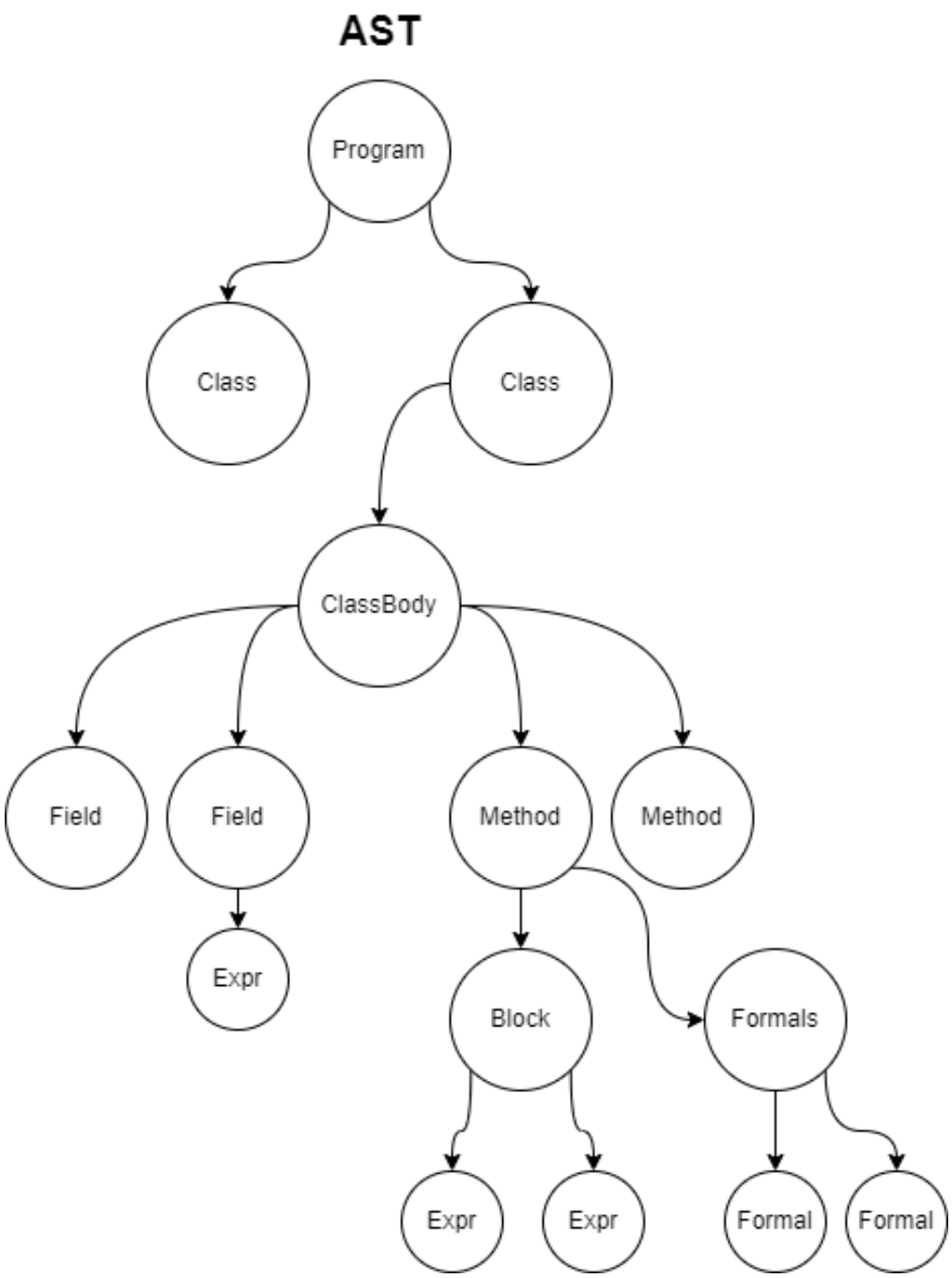
I resolved those problems using conditionally activating rules and declared exclusive start conditions.
It means that when a set of exclusive rule is activated, the lexer will only catch rules that are in that set, the others are ignored.
I used conditionally activating rules and a stack to manage the multi-lines comments.

Syntax Analysis

Again, I used the given flex/bison example to implement this part.
I created a new class 'node' that is parent of all different kind of nodes in the AST (see node.hpp).
I also created a class 'visitor' to implement the Visitor Design Pattern, it simplifies greatly the implementation to print the tree (and also the implementation the two next parts).
In order to print the AST, I used the 'visitor' class and extended it to create the class 'ParserPrinter' (implementation on ParserPrinter.cpp).

To build the AST, I modified the file parser.y to create a node for each grammar rule.
I followed the precedence rules in the VSOP manual but I still had shift/reduce conflicts.
I had to make a choice on the 'do', ':' and 'in' keywords, I chose to mark them with right precedence.

Here is an overview of my AST representation.
The expressions are not represented (If,Let,While,Assign,Unop,Binop,Call,New,Objid,Bool,Int,Str,Unit,Args).
See "node.hpp" for more details.



Semantic Analysis

For this part, I created two new visitor, 'semanticChecker' (implemented in semanticChecker.cpp) to type check the AST and 'semanticPrinter' (implemented in semanticPrinter.cpp) to print the annoted AST.

I added a variable "Type" the 'node' class to be able to retrieve the type of each node.
The class 'semanticChecker' verify everything, the inheritance cycle, redefined class,etc...
The method I used for the type checking is:
I start from the root node of the AST, then I visit the child nodes (they will set their type).
After I retrieve their type thanks to the newly added type variable and finally I do the type checking.

I made this choice because the method is simple and permit to focus on one node at a time.

Code Generation

I implemented the code generation thanks to a new 'visitor' class: 'LLVMGen' (implemented in LLVMGen.cpp).
It is inspired by the LLVM implementation example, I only used the LLVM library.
I added a variable "Value" the 'node' class to be able to retrieve the Value of each node.
I used the same method as in the semantic analysis for the code generation, I start from the root node of the AST, then I visit the child nodes (they will set their value).
After I retrieve their value thanks to the newly added value variable and finally I do the implementation.

Limitations and possible improvement

I did not encountered any problem in the 3 first parts of the project.
The part 4 is not totally finished:

- And operand do not evaluate Lhs then Rhs.
- String comparison is not working properly.
- Null values are not checked and can lead to code explosion.

Time estimation in the project

I would say I worked approximately 75 hours on the project, I did the 3 first parts last year but with a lot of bugs on the part 3.
I took time to understand the Visitor Design Pattern but it was worth it, everything became much more easier (I re-implemented part 2 to 4).