A Compiler Designed for the VSOP Language

Lycoops Vincent
Montefiore Institute
School of Applied Sciences
University of Liege
Liege, Belgium
Email: vlycoops@student.ulg.ac.be

Mathonet Gregoire
Montefiore Institute
School of Applied Sciences
University of Liege
Liege, Belgium
Email: gregoire.mathonet@student.ulg.ac.be

***Abstract*—In this report, we focus on explaining our different implementation choices, as well as our personal choices for the interpretation of a VSOP source code file.**

I. Introduction

The VSOP language is a ULg designed programming language aiming at being Turing-complete and providing the most used functionalities of well-known programming languages while focusing on keeping a lightweight syntax and little syntactic sugar.
In this context, we are building a compiler for this language that should be able to reach the LLVM transition format, to apply any other backed compiler afterwards. The designed compiler should also report errors in a convenient fashion.
We now describe each part of the frontend of the compiler in the following sections.

II. Lexer

The lexer uses a JFlex produced regular expression analyzer to cut the input file into a stream of tokens. It tries not to stop on the first error to report as many errors as possible. Literals are determined by a look-ahead of either a whitespace character or an operator, so that ending the file with a literal will ouput an error. However, as VSOP is class-defined, the last character of any file should be a closing bracket, and this should not be a problem. To ensure compatibility with tests, however, we use a stream reader that adds a whitespace to the end of the stream, allowing those matchings to occur.
Upon error, the lexer tries to determine which kind of error happened to produce more useful feedback to the user. This way, if the problems happens while parsing the beginning of a non-decimal number, the lexer will be able to output a specific message, for instance.
A final EOF token is added to the end of the stream, token used by java_cup but which is not rendered while displaying the found tokens.

III. Parser

The parser reads the stream of symbols generated by the lexer to give them as feed to the java_cup implementation of the VSOP grammar. The value associated to a mathematical expression is still not computed on the fly, even if can be known at compile time.
Each generated node of the tree is an ASTNode that still embbeds the information of position in the source code if it is a terminal symbol, or the beginning of a syntactically determined block remembered (for instance, a class, but not a parenthesis). The terminals typically have a value, either a string or an integer one, while all nodes can have properties such as position. A type is associated to the leaves of the created AST so that the type of all object identifiers is already known at this time.
The dumping is done recursively from the root node, exploring children's state.
Error recovery is kept minimal, it only tries to parsing correctly three next tokens. If this is possible, the generation of the tree is continued, but by default the generated tree is not outputted as it may refer to garbage only. The position of the error is reported to the last correctly parsed token if the symbol involved is not a terminal, or at the position of the terminal itself if known. So far, the position of elements is not recorded upwards so that some errors might still not be accurate in position.

IV. Semantics analyzer

A first uncomplete pass is performed in the AST, in order to find all the classes, and then to populate all classes with their methods. At this step, no check is done to validate inheritance nor method overriding.
From a first complete pass in prefix order in the AST, the analyzer creates the scope of each node, by adding elements for all children of a node that declares a new element in the scope. From the same pass, the semantic analyzer creates a tree of classes that extend from each other and verifies that each atom used as a leaf (not in a declaration) is indeed present in the scope of this leaf. A second pass is then issued in postfix order, to add a type information to each statement and block of code. The leaves' type is already known from the parser, and the type of a supernode is uniquely determined by the operation applied to children expression of this operation. At the same time, the analyzer

verifies that the involved operations exists, are correctly called, with arguments matching in type and number. Thanks to the ”extends tree”, the analyzer is able to substitute a child class to a parent one to retrieve a legible operation.

V. LLVM IR GENERATION

Section still under work.

The generation comes down to building a OOP-free representation of an OOP language: each method is turned into a global function whose name is prefixed with the name of its class, and that takes as first argument a structure representing the class: a structure that contains but the fields of the class.

The new operator invokes the constructor of the class, that issues a ”super();”-like call if it extends something else than Object. The fields in a class are ordered with highest in extends-tree first so that those calls are easily managed.

The code is then emitted recursively, and the entry point of the program is set to a new Main.main().

VI. CONCLUSION

We presented our still under work compiler for the VSOP language. As one can see, it boils down to assemble the right generators finely tuned for a given language, but generic options to generate compilers now exist.

ACKNOWLEDGMENT

The authors would like to thank the developers of the tools we used to build this compiler, namely JFlex, java_cup and LLVM so far.

REFERENCES

[1] G. Klein, *JFlex*, http://jflex.de/

[2] Technical University Munich, *java_cup*, http://www2.cs.tum.edu/projects/cup/

[3] LLVM, *LLVM IR specification*, http://llvm.org/docs/LangRef.html