Group 0 - Final Report

INFO085 - Compiler

Develop a compiler for the VSOP language

Thibaut Chavet Maxim Henry Maxime Massart

May 26, 2017

Contents

1 Introduction

This project is done for the compiler course given by Prof. Geurts during the academic year 2016-2017. We are asked to implement a vsop compiler. The vsop language is defined on the course website.

As defined in the assignment, `The Very Simple Object-oriented Programming` language (or VSOP) is, as its name implies, a simple object-oriented language.

2 Our compiler

We chose the C++ 11 language to implement our vsop compiler because it is well known by the team and allows us to use powerful tools such as GNU Bison and GNU Flex.

The implementation was divided into 4 main steps:

- the Lexical Analysis (cf 2.1)
- the Syntax Analysis (cf 2.2)
- the Semantic Checking (cf 2.3)
- the Code Generation (cf 2.4)

Status of the compiler

Our compiler generates LLVM code which is then compiled to machine code using llvm compiler version 3.5.

2.1 Lexical Analysis

The lexical analysis is used to make sure the words inside the code to be compiled are allowed by the language, and parse the vsop file. The goal of the scanner we are then able to generate, is to create tokens to be translated into tree nodes at the next steps. We generate the tokens using the GNU Flex program. Flex will generate a .c and a .h files which will be compiled and do the parsin.

From the moment, we implemented the syntax analysis, the yylex() function is called from the yyparse() which is used in the second step. It has the drawback of not allowing the simple tests performed in the first step as both errors coming from Lexical and Syntax analysis will be displayed as the file is analyzed.

We had an issue with this lexical part. As our C++ Bison yyparse() calls yylex itself, when the input of this part is not correct in the vsop formalism, all the tokens are not read until the end, but only until the syntax error. We thought about a simple solution which was to do the first phase(GNU FLex) until the end. Then launch do the second phase(GNU Bison) which would also redo the first one. It is not that efficient after all but it would be useful to pass your automated tests.

2.1.1 Tokens

The tokens that we defined are explained in Table 1.

Those tokens will be used by GNU Bison.

2.2 Syntax Analysis

To build a parser for our language, we used GNU Bison

The parser is built from a grammar that is defined in the assignment.

Each function is contained in the class. Each block contained in a function has as parent that particular function.

For this part, we succeeded in all automated tests.

2.2.1 Building the tree

The tree is composed of classes inheriting from AstNodes classes. The root node is a ProgramNode which contains a ClassNode for each class. Each AstNode has a pointer to their parent, and to their children. We first used a single AstNode class, with an attribute indicating what type of node it was, but while that worked for syntax analysis part, we soon realized that it was not flexible enough for the next parts. So each type of node has functions that are needed for any node, such as `printTree` (print the syntactic tree) `getType` (for semantic analysis) and `llvm` (generate the llvm code for that node, but also specific functions depending on what node it is (`getMethods` for the classes for example).

This tree will be used differently in the following steps.

Each class is defined in a file given in the `nodes/` directory.

2.3 Semantic checking

Semantic checking is done through the `Semantic` class.
We also use the `Types` class, which defines a hash table that maps the type (primitive type such as `int32` or types defined through classes), and the ClassNode associated to it (or null pointer for primitive types), and contains functions to make operations on that table. This table is useful to know if this type has already been declared in the next class that will be analyzed, to know if all classes extending others are all declared, and later to get the `ClassNode` of a type.

2.3.1 Class checking

This part checks:

- Class redefinition
- Parent definition
- Inheritance cycles

Token	Description
INT_LIT	Represents a integer literal
AND	Represents the and boolean operator
BOOL	Represents the type definition of boolean
CLASS	Represents the class definition keyword
DO	Represents the beginning of a while repeated routine
ELSE	Represents the else keyword in the if statement
EXTENDS	Represents the inheritance keyword performed in class definition
FALSE	Represents the false boolean literal
IF	Represents the if statement keyword
IN	Represents the in keyword used in the let statement
INT32	Represents the type definition of an integer on 32 bits
ISNULL	Represents the isnull conditional keyword
LET	Represents the definition of local variables
NEW	Represents the object instantiation keyword
NOT	Represents the negation in conditions keyword
STRING	Represents the keyword defining a string element
THEN	Represents the keyword introducing the affirmative part of the condition contained in the if statement
TRUE	Represents the true boolean literal
UNIT	Represents the unit type (similar to void in C)
WHILE	Represents the loop keyword
<strval> TYPE_ID	Represents a Type identifier
<strval> OBJECT_ID	Represents an object identifier
<strval> STRING_LIT	Represents a string literal
LBRACE	Represents the beginning of a block
RBRACE	Represents the end of a block
LPAR	Represents the beginning of a parenthesis used in function calls and functions declaration
RPAR	Represents the end of a parenthesis
COLON	Represents the colon symbols delimiting type definitions
SEMICOLON	Represents the semi colon symbol at the end of a non-returned function instruction
COMMA	Represents the comma symbol to separate parts
PLUS	Represents the plus symbol to perform the addition
MINUS	Represents the minus symbol to perform the subtraction
TIMES	Represents the star symbol to perform multiplications
DIV	Represents the division symbol to perform divisions
POW	Represents the power symbol to perform the power
DOT	Represents the dot symbol used for accessing object elements
EQUAL	Represents the equal symbol used in conditional comparison
LOWER	Represents the lower symbol used in conditional comparison
LOWER_EQ	Represents lower or equal symbol used in conditional comparisons
ASSIGN	Represents the local variables assignments

Table 1: List of Tokens

- Main class and method prototype existence

Inheritance cycles checking is done by looking if the parent of class A is not an ancestor of class A. This works to check that A is not in a cycle, but if the parent of A is in a cycle, it will recursively call the parent, until a stack overflow occurs, so that it a flaw of our compiler.

2.3.2 Scope checking

During this part we check everything else, mainly if the types agree, and if variables, fields and methods used are defined.
This is done by going through the tree and performing semantic checking on each node, according to what type of node it is. To display the errors, we contained that into the `Semantic` class, for more flexibility. All errors that occur are passed along using a vector. The errors are represented using the `SemErr` structure: it contains the line and column where the error occurred, and the error message to be displayed.

2.3.3 Types retrieving and checking

The second time we go through the tree, we needed mechanisms to know the closest common parent that we use to determine the output of a dynamic instantiation. This step has the purpose of checking that all types are correctly implemented, and that all expressions have the correct type. This is done using the existing tree of nodes, instead of building a separate scope tree.

2.3.4 Redefinition

We have to determine if the fields are not redefined. The methods can be overwritten if they agree to certain conditions. We implemented this by going through the nodes and asking the parent classes (if needed) to find out if it is a redefinition. We also check the arguments of the methods during this step.

2.4 Code Generation

We learned that a LLVM library existed. However, it was quite difficult to handle so we decided to develop our own version LlvmManager. At first, we were using a llvm tutorial which had the version 4 or 3.8. However, we realized we needed to use version 3.5. This is why we used

2.4.1 How it works

First, the LLVM will generate the header of the code. It writes down the methods types declarations.

Then, the LLVM will generate the class declaration.

After that it declares the Main class and allocates the methods variables.

It finishes by going through the tree to get the llvm translation of each node and it finishes by declaring the strings as global variables.

2.4.2 Choices and / or constraints

- We saw that when we try to use %0, llvm wants to use it for its own purpose (maybe in version 4). So we chose to begin giving names to our temporary variables beginning to %1.
- We first implemented the instantiation of a new object as a block of operations allocating and initializing the objects. At that time, we weren't really aware of the technique of using a knit method to do so. We then decided not to implement it since the main purpose of llvm code is not to have the greatest readibility. The only drawback of keeping or first idea is just making the llvm file a bit heavier
- We saw that getelemntptr in LLVM requires a load just afterwards. So we did it every time.
- The functions always takes as first argument the element on which they act. It allows us to use the keyword 'self' accordingly.
- The number of the variables does not start back to 0 when the program changes function. We first thought that it was module dependant but we decided for debugging and code understanding not to reset that counter variable.
- We realized that the declare statements in llvm language can refer to functions declared in other modules. We then used them at the end of our code.
- When calling a function, the arguments are given as values and not as references. This has some drawbacks but we had to make up our choice.

3 Code structure

This section is used to explain the structure of the code and its purpose.

3.1 Doxygen

We were able to generate Doxygen files. You are able to generate it using `make doc`. This command also displays the documentation directly in Firefox. Please have a look at `../doxygen/html/index.html` from the `report/` folder.

3.2 Folders

The table 2 reflects the folders distribution.

Folder	Description
.	Default folder containing the files for Flex, GNU Bison and LLvmManager class
doxygen/	The documentation (please see subsection 3.1)
nodes/	The classes used to build the tree (cf. 2.2.1)
report/	Contains the report pdf
semantic/	Contains the Semantic class and the different declarations required in step 2.3
vsopl/	Embeds the additional elements for code generation

Table 2: Folders descriptions

4 Improvements

- Add an initial function : Init. It will make program heavier because it requires saving registries, ... due to yet another function call. We had to know if the LLVM compiler would optimize it so that we do not do something that will be removed just after.
- We have a problem with the inheritance of classes cycling. We tried to limit this problematic but we need to find a more elegant solution.

5 Conclusion

We know that our implementation is not perfect. In section 4, you could find the improvements that we made.