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 to parse the vsop file. The goal of the scanner that is then generated, is to create tokens for the grammar, which will create 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 parsing.

The yylex() function is called from yyparse() which is used in the second step, so it has the drawback of not allowing the automatic tests performed in the first step to go through if there are syntax errors. Errors coming from lexical analysis will be displayed as the file is parsed using the grammar.
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 the 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. At each definition in the grammar, the appropriate node is created, using the previously created nodes as children, and returning the created node for the next grammar entry, as it is performed top-down.

2.2.1 Building the tree

The tree is composed of classes inheriting from the AstNode class. 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. At first we used a single AstNode class, with an attribute indicating what type of node it was, but while that worked for the 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 all nodes, 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 ClassNodes for example).
This tree will be used differently in the following steps.

2.3 Semantic checking

Semantic checking is done through the `Semantic` class.

2.3.1 Class checking

This part checks:

- Class redefinition
- Parent definition
- Inheritance cycles
- Main class and method prototype existence

We also use the `Types` class, which defines a hash table that maps the type (primitive types 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 a type has already been declared (class redefinition), to know if all classes extending others are all declared (parent definition), and later to get the `ClassNode` of a type, when we only have a string of the type name.

Token	Description
INT_LIT	Represents a number
AND	Represents the and keyword
BOOL	Represents the bool keyword
CLASS	Represents the class keyword
DO	Represents the do keyword
ELSE	Represents the else keyword
EXTENDS	Represents the extends keyword
FALSE	Represents the false keyword
IF	Represents the if keyword
IN	Represents the in keyword
INT32	Represents the int32 keyword
ISNULL	Represents the isnull keyword
LET	Represents the let keyword
NEW	Represents the new keyword
NOT	Represents the not keyword
STRING	Represents the string keyword
THEN	Represents the then keyword
TRUE	Represents the true keyword
UNIT	Represents the unit keyword
WHILE	Represents the while keyword
TYPE_ID	Represents a Type identifier (string beginning with a capital letter)
OBJECT_ID	Represents an object identifier (string beginning with a lowercase letter)
STRING_LIT	Represents a string literal (text between quotes)
LBRACE	Represents an opening bracket
RBRACE	Represents a closing bracket
LPAR	Represents an oping parenthesis
RPAR	Represents a closing parenthesis
COLON	Represents the colon symbols
SEMICOLON	Represents the semi colon symbol
COMMA	Represents the comma symbol
PLUS	Represents the plus symbol
MINUS	Represents the minus symbol
TIMES	Represents the star symbol
DIV	Represents the division symbol
POW	Represents the power symbol
DOT	Represents the dot symbol
EQUAL	Represents the equal symbol
LOWER	Represents the lower symbol
LOWER_EQ	Represents lower or equal symbol
ASSIGN	Represents the -¿ symbol

Table 1: List of Tokens

Inheritance cycles checking is done by looking if the parent of class A is not a descendant 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 Methods

Semantic checking for the methods is done by checking that:

- The return type is valid (primitive type or defined using a class), this is done using the `Types` class.
- The method has not already been defined
- Parameters checking:
  - The name of the parameter is not 'self'
  - The type is valid
  - There are no other parameters with the same name
- The return type of the body of the method is the same as the method's return type

2.3.4 Fields

- The field has not already been defined
- The type is valid
- If there is an initialization expression, its type is the same as the field's type.

2.3.5 Expressions

Semantic checking for the expressions is done using the `getType` method, which computes the type of the expression (and of the subexpressions) and checks for semantic errors. It returns an `ExprType` structure, which contains the computed type of the expression, a boolean indicating if an error occurred, and the vector of `SemErr`. It tries to check as much as it can even if there is an error, in order to report as many errors as possible.
When checking if the types agree, it uses the `isA` method of `ClassNode` to check if a class

inherits from another, as well as the `commonAncestor` method to find the common ancestor of two classes, for example to determine the type of an if statement.

2.3.6 Scope table

We first thought of using a scope table to store the variables defined in the current scope, to check if each used variable is defined, but we realized that it would add too much complexity, and that it is easier to simply go up the tree and look if we can find a node that defines that variable.

2.3.7 Vsopl

Vsopl is the vsop library that contains the built in classes and methods, such as IO. We create nodes for the classes and the methods, that contain nothing in them, their implementation is directly done in llvm. Those nodes are not contained in the main tree, but still exist so that their functionalities can be accessed (for example a class that inherits from IO can get its parent class node and have access to the methods defined by IO).

2.4 Code Generation

We learned that a LLVM library existed. However, it was quite difficult to handle so we decided to directly generate the llvm code ourselves. It provides more flexibility
At first, we were using llvm 4, however we realized we needed to use version 3.5.

2.4.1 Classes

The classes are implemented in llvm using a structure that contains the methods vector, and the fields of the class. We get the fields by going through the class, then putting them into a hash table, and then do the same for the class parent, and so on. If a field is redefined in a parent class, it is already in the table, and won't be added again. They are given a number, which is their order in the structure, that way we can access any field in the structure later, by looking up their place using their name.
The same is done for the methods, except that we do not put them directly in the class structure, but in a separate one, which will then be placed at the beginning of the structure.

2.4.2 Vsopl

The llvm code of the vsop library is in the file `vsopl.ll`. We mainly generated it using clang.

2.4.3 LLvmManager

This class is used to do the common llvm operations, such as getting a pointer to a method, or writing llvm code and get a unique variable name. When instantiating it, multiple output streams can be given, that way it can write the llvm output code to the

screen and in a file at the same time. This manager is passed along each node during llvm generation.

2.4.4 Header

First we generate the header of the code, which declares all the types that we use, as well as declarations of the vsopl methods, which are defined in another file.

2.4.5 Main

The llvm main class instantiates the vsop Main class, then calls the main method.

2.4.6 Llvm code generation

The llvm code is generated by going through the tree and generating the appropriate llvm code for that node, usually by first generating the code for the sub node, and using the result.
That function returns a string which is the name of the llvm variable in which the result of the expression is contained.
Once again, we did not use a table to contain all the currently defined variables, but simply go up the tree until we get to the node that defines the variable (using the method `getLlvmVariable`). At first we used simple llvm variables for each vsop variable, but we realized that it would not work for loops for example, where at each loop the same statements are called, therefore they would always do the same thing, unless we used pointers. So now each vsop variable is represented using a llvm pointer, so that when a llvm statement is executed multiple times, the value pointed by the variable changes, instead of just redefining the same variable using the same statement.

2.4.7 strings

For strings it is much easier to define them as constant than to allocate them in the middle of the code. But we are only aware of the existence of a string while we are already in a function, so it cannot be declared as a constant there, and declaring them at the beginning would require scanning the whole tree just to get the strings. So what we did is to put the string in a vector managed by `LlvmManager`, and declare all the strings at the end of the file.

2.4.8 Choices and/or constraints

- At first we wanted to use %*number* for unnamed variables, but there were too many constraint in llvm for that (requiring that we always start at 1 for each new function for example), so instead we chose to use %.*number*.
- We implemented the instantiation of a new object as a block of operations allocating and initializing the objects directly in place in the llvm code, instead of a separate method to do so. It does not really change llvm performance, it only makes the file a little less readable.

- The functions always take as first argument a pointer to the object to which they belong, to be able to access the fields and methods of that object.
- When calling a function, the arguments are given as values and not as pointers. This was simply a choice we had to make, but seemed more practical when using vsop.
- At first we implemented the if instructions using the llvm phi instruction, but it caused problems for nested ifs, because in that case the previous label we branched from was not the right one. So we chose to use pointers instead.

3 Code structure

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

3.1 Doxygen

We commented the code using doxygen format. The doxygen documentation can be generated using `make doc`. This command also displays the documentation directly in Firefox.

3.2 Folders

The table 2 describes each folder of the project.

Folder	Description
.	Root folder containing the files for Flex, GNU Bison and the LLvmManager class
doxygen/	The documentation (see subsection 3.1)
nodes/	The classes used to build the tree (cf. 2.2.1)
report/	Contains the report
semantic/	Contains the classes used for semantic analysis 2.3
vsopl/	Contains the required class for the vsop library as well as the llvm code of the library

Table 2: Folders descriptions

4 Improvements

- Add a function for class allocation. Right now it is done in the middle of the llvm code, where we need it. The llvm code would be more readable if it is done in a separate function.
- Correct the inheritance cycles problem mentioned earlier.

5 Conclusion

We have a basic compiler that, for all we are aware, can compile vsop code, but some problems and bugs could still exist for cases we did not test. It could be improved for optimization, but would required more knowledge of how llvm works and how the code needs to be structured to allow more llvm optimizations. We could also easily add extensions, due to the modularity of the code. It would only require adding some tokens, grammar rules, and nodes for which we would write their semantic checking and llvm functions.