Faculty of Information Technology
Brno University of Technology

Compiler Construction (in English)

VYPa19 compiler

2019/2020

Extensions: none
Tomáš Aubrecht (xaubre02): 50 %
Jan Kotráš (xkotra01): 50 %

December 22, 2019

Contents

1 Introduction

Our assignment was to study the specification of a simple programming language VYPlanguage inspired in C language and Java with the support of the object–oriented paradigm including objects, classes, polymorphism and simple inheritance. After that, we had to create and define its grammar and build a compiler using the syntax-directed translation. This compiler translates source code written in VYPlanguage into VYPcode, which is an intermediate code similar to assembly language.

We were allowed to use tools for parser generation such as Flex/Bison, ANTLR or PLY and the choice of the language in which we implement the compiler was up to us. Only restriction was that this language has to be usable on Merlin server. The VYPa code interpreter had been already implemented, so it was not our goal to develop it. The interpreter was available as a part of the project specification.

2 Compiler structure

Since we are both experienced with the Python programming language and it was a possible choice, we decided to write the compiler in that language. In this section we will describe the individual parts of the compiler as we gradually developed it.

First we designed the compiler architecture, where each part of a source program in VYPa19 language is represented with a corresponding class in Python. When we had parsed a source code to our object representation, it was much easier to work with. The architecture is described below in the subsection 2.1. Next subsection describes the generation of the lexer and parser of VYPa19 language. On top of that we implemented the semantic analysis described in the subsection 2.3 on page 4 and code generation described in the subsection 2.4 on page 4.

2.1 Architecture

The compiler is represented by the class `Compiler`, which is initialized and manipulated with in the main script. The most important method of this class is the method `compile()`, which directs the whole compilation process. We implemented a system of exceptions, where each exception has its own compilation error code corresponding to the type of an error that occurred. Base exception is `CompilationException`, which is caught in the `compile()` method and from which every other exception inherits. These exceptions are:

- `InternalErrorException`
- `LexicalErrorException`
- `SyntacticErrorException`
- `SemanticErrorException`

In the exception `SemanticErrorException` it is possible to specify the error code, because a semantic or type error can occur during the semantic analysis.

There is an abstract class `Entity` that holds a name of an entity and line and column of itself in a VYPa19 source file. This entity could be a variable, function or class and the program consists of these entities. Therefore, there are classes `Variable`, `Function` and `Class` that implements this abstract class and represents the main components of the program. There are also other classes that complements these main classes, e.g. `Statement` or `Expression`.

2.2 Lexer and Parser

For generating a lexer and a parser we chose ANTLR. ANTLR stands for *ANother Tool for Language Recognition* and it is a powerful parser generator. We create a grammar and from that grammar, ANTLR generates a parser that can build and walk parse trees. ANTLR introduces LL(*) strategy as part of the top-down parsing approach. This tool is made up of the tool, used to generate the lexer and parser, and the runtime environment needed to run them. It's a Java program, but it can be used with many others languages, in our case - with Python.

An ANTLR grammar consists of two main parts: parser rules and lexer rules. Parser rules are used for the definition of a language syntax and lexer rules are used for the definition of lexemes. Lexer rules are at the end of the grammar file and in our project we used rules fragments, which are reusable building blocks for lexer rules, e.g. *LETTER* or *DIGIT*.

The resulting grammar can be viewed in the file `VYPa19.g4` in the root directory of the project. To generate the lexer and parser, we run the command `<antlr4> -Dlanguage=Python3 VYPa19.g4`, where `<antlr4>` specifies the path to the ANTLR tool. This command is a part of the Makefile and it generates our lexer and parser in Python source code, which we simply included in our project.

2.3 Semantic analysis

Thanks to the generated lexer and parser, we simply go through a source code of the VYPa19 program and store the definitions of functions and classes. We store them in the global symbol table. Compiler's symbol table is represented with another important class – the `SymbolTable` class. This class is simply a wrapper for the standard Python dictionary, where its keys are symbols and values are instances of program entities. Each symbol table, except the global one, has a parent symbol table. This approach creates a scope hierarchy for each block of code. For example, function `main()` is stored as an instance of the class `Function` under the key *main*. This is done in the first pass.

Now we have acquired signatures of these functions and classes and in the second pass, we initialize them – we analyse their bodies. We go through each statement of each function and class and check its semantic aspects. If something does not correspond to the language specification, we raise a corresponding exception.

2.4 Code generation

For code generation exist two classes *CodeGenerator* and *CodeCreator*. *CodeGenerator* is a class and one instance that manage code generation. First of generation the generator allocate variable, create register labels and some control labels. After that the generator start to generate built in functions, implemented function, classes, methods,...

Every semantic class of the python code contain method, named *code* or *codecreate*. The method generate specific code assigned semantic element. For generation code the method use creator - *CodeCreator*. The creator exist as sigleton, and it is designed to help commands creation - for every target command the creator has method. Creator save generated code inside self. That code is save in the last part of compilation.

2.5 How to compile

To translate a VYPa19 source program into VYPcode, you need to generate the lexer and parser using `make` command and then you can execute the `vypcomp` file located in the root directory. It is a simple bash script that executes the main Python file of our compiler. It takes 2 positional arguments, where the first one specifies the input source file and the second one is optional and specifies the output file. Default output filename is `out.vc`.

3 Development

For development the our team use two different platforms Windows and Linux. During the development, there is no problems with platform incompatibility. Whole team use for python development the PyCharm. Developed code is under version control software. For that purpoase we use git and BitBucket like remote repository.

3.1 Work division

Teams consisted of two members and we both did the same amount of work on the project.
More specifically:

Tomáš Aubrecht

- lexical, syntactic and semantic analysis
- testing
- documentation

Jan Kontráš

- code generation
- documentation

3.2 Communication and tools

As members of the team, we did not know each other personally, so we had to choose a communication platform. We decided to use a social network and we both had a Facebook account, so it was a simple decision. Using its chat, we planned our meetings, which took place at the faculty before or after the lectures.

We were developing on the Windows operating system and for tracking and review our work, we used the version control system Git. Specifically, we created a private repository on GitHub that was only visible to us. This system solved issues with source code inconsistency by providing us with the merging tools that fix it.

3.3 Testing

Because we knew it would be necessary to run a larger set of tests, we created a script that automatically runs and evaluates them. Thanks to the script, it was possible to perform a single command check instead of running individual tests, which was very important to ease our work. For this purpose, we decided to use Python's package *unittest* to do the work. We created a separate unit for each part of our compiler, each consisting of several tests that checked the given problems. Main test units were: lexical errors, syntactic errors, semantic errors, code generation errors and valid programs.

4 References

Faculty logo: https://www.vutbr.cz/data_storage/multimedia/jvs/loga/02_fakulty/FIT/1-zakladni/EN/PNG/FIT_color_RGB_EN.png

ANTLR tool: https://www.antlr.org/ and https://tomassetti.me/antlr-mega-tutorial/

Literature: VYPa lectures