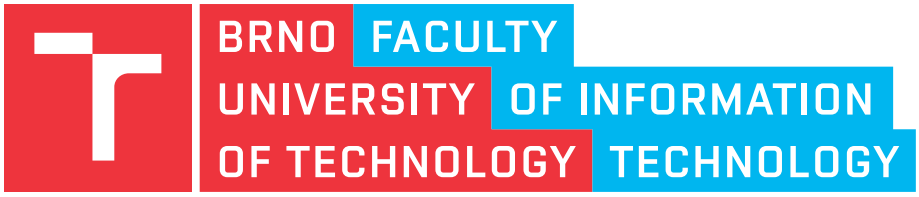
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGYÍ



Compiler Implementation for VYPe16 Programming Language

VYPe - Compiler Construction

Filip Benna, xbenna01
Tomáš Bruckner, xbruck02

2016

1 Introduction

This documentation is describing the implementation of the compiler for VYPe16 programming language. It is a project for the course VYPe Compiler Construction is taught at Faculty of Information Technology at Brno University of Technology. VYPe16 is a limited version of C language. The source code is compiled to the MIPS32 instruction set. Full specification can be found at

2 Front-end implementation

The front-end of the compiler is doing analysis of the given source code. It checks for lexical errors, syntactic errors and semantic errors. During the analysis, it creates parse trees that are used as the input for the back-end part of the compiler. For easier implementation of the front-end, ANTLR4 framework has been used. In this framework, tokens and grammar rules are defined and it generates Java classes and interfaces, that are used for the scanner and the parser.

2.1 Lexical analysis

For the lexical analysis, a lexer grammar has been defined in the file *VYPeLexer*. The lexer grammar is basically a set of the rules (tokens) defined by something similiar to the regular expressions. Some rules has been defined, so that white spaces, line comments and block comments are skipped. There was an option to send them to different channel (HIDDEN channel is recommended as best practice) instead of skipping them, but there was no use for them later, so it was decided to skip them.

The rules of the lexer grammar are used by ANTLR4 to generate VYPeLexer class, that is used for creating a token input stream. This token stream is later used as input for the parser class described below 2.2.

One special rule (token) has been defined that is not in a specification, *Error* token. ANTLR4 has not got the best support for detecting lexical errors. The way it should be detected is using *DefaultErrorStrategy* class, but it has some problems, so it was decided that we will create special *Error* token, that catches every invalid tokens. It is later checked if any *Error* token has been encountered, in which case the program ends with a lexical error code.

2.2 Syntactic analysis

For the syntactic analysis, grammar rules has been defined in the file *VYPeParser*. Unfortunately, ANTLR4 cannot handle *EOF* token in the first rule, so there is a special rule *parse*, that just takes one alternative rule *start*, which is the start rule for the grammar.

Based on the grammar rules, ANTLR4 generates parser class, that parses input token stream. The parser rule *start* is checked, because it is the starting rule for VYPe16 language. In case of the syntactic error, the program ends with a syntactic error code. It can be easily checked, because the parser class stores the number of the syntactic errors in its instance.

There was also a problem with defining operators with same priority. For different priority, it is quite easy. They just have to be defined in an order from the highest priority to the lowest. For same priority, the dynamic rule has been used. It specified alternatives with the same priority. It could also be defined, but this is clearer solution.

Labels for rule alternatives were sometimes used, because it made semantic analysis easier. Visitor/listener class takes all labels and creates parsing methods for them. The rules are specified in the grammar below 2.4.

2.3 Semantic analysis

For the semantic analysis, visitor and listener classes were used. *FunctionTable* class is used for tracking what functions are already declared or defined. *SymbolTable* class is used for tracking

all variables in current scope. For the nested scopes, every symbol table can have a reference to its parent scope (symbol table).

There was also a problem, that there had to be some way, how to track type of current expression. This problem was resolved by *Value* interface and a few implementations of the interface for each data type. These classes are also used when a new function is declared/defined. They are used for the parameters declaration/definition. There was a problem with embedded function *print*, because it has variable number of parameters. It was solved by defining *VariableValue* class that implements *Value* interface and represents, that the function has variable number of parameters. For user defined functions, there are listeners that parses all parameters and stores them in function table.

Semantic check takes function declarations and definitions sequentially and looks into function table if there is no other declaration/definition of the same function. In case there is, throws *SemanticException*. If it is a function definition, the body of the function is checked. No variables cannot be used if it was not declared before. Symbol table is used for this case. Similarly, there cannot be any function calls if the function has not been already declared (function table is used). Every expression type is checked. It has to correspond to expected type. For instance, if there is an addition of a number and function call that returns string type, *SemanticException* is thrown.

2.4 Grammar

TODO ADD GRAMMAR!!!!!!!!!!

3 Back-end implementation

design, implementation, intermediate
code, symbol table, special techniques used, optimizations, algorithms

4 Literature

Literature used, including citations of any non-original materials (pictures, statistics etc.).