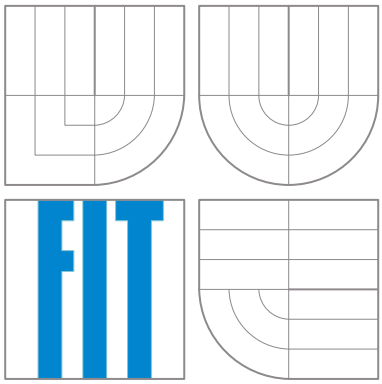
Faculty of Information Technology
Brno University of Technology



Documentation for VYPe

Vendula Poncová `xponco00` 75%
Martin Šifra `xsifra00` 25%

August 22, 2016

1 Introduction

In this documentation, we describe the design and implementation of the compiler of the programming language `VYPe2014`.

2 Lexer and parser

As the front part of our compiler, we have used tools `flex 2.5.35` and `bison 2.5`. We were going to implement the compiler in `C++`, thus we have worked with `C++` interface in `bison`. However, `flex` does not support `C++` very well, so we have used `C` interface. The current version of bison is `3.0.2`, but `merlin` does not support it.

Lexical analyser is described in file `lexer.l`, where we define the regular expressions of lexems for `VYPe2014`. The most challenging were expressions for string and character literals and skipping the comment. For comments, we use state `COMMENT`, that allows us to ignore characters until we reach the end of the comment. For string and character literals, we explicitly enumerate allowed `ASCII` codes and escape characters.

Syntax analyser is defined in file `parser.yy`, where there are definitions of tokens and grammar rules with actions. We have implemented no extension, thus it was not difficult to design the grammar. The description of the grammar can be found at the page 4. We skiped the rules for expression.

3 Compiler and generator

3.1 Symbol table

Our symbol table consists of a table of functions and tables of variables. Every function holds a list of tables of local variables. In this way, we can deal with overlapping variable names in nested blocks of statements. We use an auxiliary stack, where we push or pop the pointers to variable tables, when we enter or leave a block of statements. When we need to check if the variable was already defined, we search the tables in the stack from top to down. If there is a variable that overlaps with another variable, we will find the one in the most nested block.

Variables in a variable table hold their name, type and sometimes their value. Functions holds their identifier, list of variables, that are parameters, instructions of three-address code and two flags. One flag symbolizes that the function was declared but not defined, second flag symbolizes, that the function is built-in.

We use the symbol table for semantic analyses. After the semantic error, we always recover and continue with the compilation.

3.2 Three-address code

In the syntax analyser, we create three-address code from parse tree. The code is represented by a list of simple instructions. We use several types of instructions. Instruction label represents a place in a code, where we can jump. Instruction expression represents an arithmetical or logical operation between two variables, where the result of the operation is stored in the third variable. For these result variables we generate temporal variables. Cast instruction represents a cast of the variable to another type. There are mentioned three ways of casting in the assignment, however, in the example is also a cast from integer to integer, so we accept a cast to same type as the type of variable as well. Load instruction allows to save a constant value into a variable. Assignment instruction assigns a value from one variable to another. Jump instruction jumps to a label, jump-if-false instruction jumps to label, if the variable is false. Call instruction calls a function with some parameters and stores the result into a temporal variable. Return instruction returns a value of a variable and exits the function.

3.3 Generating assembly

Assembly is generated from the tree-address code. We had to implement our own stack, so we can call functions. The target code starts at the function main, pushes all local variables of main on the stack and runs the instructions of the main function. If another function is called, we push the parameters, the value of instruction pointer and the value of context pointer on the stack, set the context pointer on the top of stack and jump to the function. The function pushed all local variables on the stack and set parameters with the values on the stack. When the function returns, it sets the top of the stack to the value of the current context pointer, restores the context pointer and instruction pointer, pushes the result on the stack and jumps back to the caller.

One of the problems of generating target code was storing strings. However, we have noticed, that string cannot be ever modified. Therefore, we can store string variables as pointers to strings and when we assign one string variable to another, we just set the pointer. Variables set to string literal can just point to a constant defined in a memory. String parameters can be passed by reference. The only exception is the return value of the called function. When the function returns string, it has to be copied to the stack, so it will not be lost, when we pop the context.

3.4 Implementation

We have implemented the compiler in `C++`.

Errors are handled in a file `error.cpp`, where there are defined macros for errors and debugging. The first error code is stored in a static variable and returned as the return value of our program.

Symbol table is implemented in a file `symtable.cpp`. There are defined classes `Variable`, `VariableTable`, `Function`, `FunctionTable` and `SymbolTable` and two enums `Type` and `Operation`. The variable and function tables are implemented as maps from the library `std`.

Classes representing the three-address instructions are defined in a file `instruction.cpp`. All instructions inherits from class `Instruction`. Every instruction has implemented a method `generate`, that returns a string with generated assembly code.

The parsing is driven by a class `Driver` from file `driver.cpp`. Driver runs the parsing, generate the three address code and works with a symbol table. The instance of this class is a parameter of the parser and the lexer, thus they can use methods from this class.

If the parsing is finished with success, we call the `Generator` from file `generator.cpp`, that accepts the output file and a symbol table of functions and generates assembly. As we mentioned before, most of the generating is defined in a file with instructions.

The program starts in a file `vype.cpp`. It processes parameters, runs driver and generator and returns a return value.

4 Running the program

The program can be compiled by command `make`. It creats the file `vype` as required.

We have written a testing script in `Python 3.2` and prepared about 80 tests in a folder `test`. The command `make test` will run all the tests and show the results.

5 Division of work

Vendula Poncová	lexical analyser syntax analyser three-address code compilation driver documentation tests
Martin Šifra	code generator

6 Summary

We have implemented the compiler for the programming language `VYPe2014`. For lexical and syntax analysis, we have used `flex` and `bison`. Compiler and generator are implemented in `C++`.

A Grammar

G = (N, T, P, S), kde:

N = { <program>, <functions>, <fce_declaration>, <fce_definition>, <stmt_list>, <stmt>, <type>, <datatype>,<datatype_list>, <id_list>, <param_list>, <argument_list>, <expr_list>}

T = { (,), {, }, *;* , id, , , =, if, else, while, return, void, int, char, string }

S = <program>

P = {

<program>	⟶	<functions>
<functions>	⟶	ϵ
	|	<fce_declaration><functions>
	|	<fce_definition><functions>
<fce_declaration>	⟶	<type>id (void) ;
	|	<type>id (<datatype_list>) ;
<fce_definition>	⟶	<type>id (void) {<stmt_list>}
	|	<type>id (<param_list>) {<stmt_list>}
<stmt_list>	⟶	ϵ
	|	{<stmt_list>}
	|	<stmt><stmt_list>
<stmt>	⟶	<datatype><id_list>;
	|	id = expr;
	|	id (<argument_list>);
	|	if (expr) {<stmt_list>} else {<stmt_list>}
	|	while (expr) {<stmt_list>}
	|	return expr;
	|	return;
<type>	⟶	void
	|	<datatype>
<datatype>	⟶	int
	|	char
	|	string
<datatype_list>	⟶	<datatype>
	|	<datatype>, <datatype_list>
<id_list>	⟶	id
	|	id, <id_list>
<param_list>	⟶	<datatype>id
	|	<datatype>id, <param_list>
<argument_list>	⟶	ϵ
	|	<expr_list>
<expr_list>	⟶	expr
	|	expr, <expr_list>

}