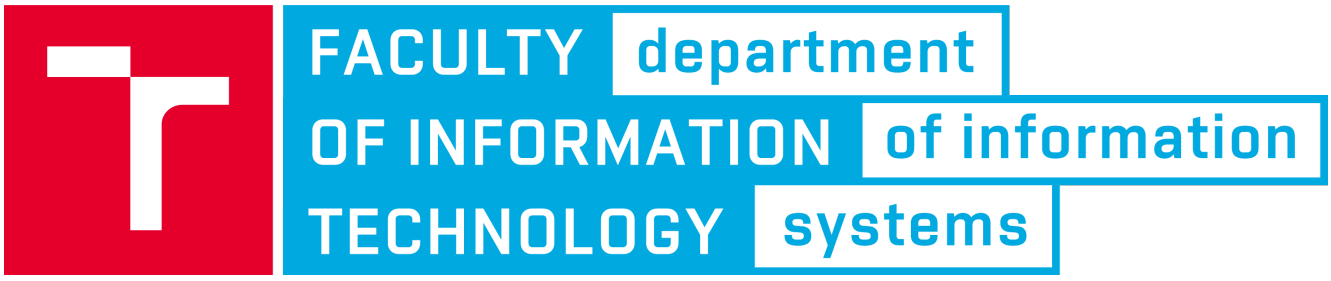
Faculty of Information Technology
Brno University of Technology



Documentation for VYPe 2015

Michal Danko `xdanko02` XX%
Martin Šifra `xsifra00` YY%

January 1, 2016

1 Introduction

The documentation describes design and implementation of our solution of the compiler of VYPe2015 programming language for processor MIPS32-Lissom.

2 Lexer and parser

To save time and work we've used tools flex as a lexical analyser generator and GNU Bison as a parser generator. Everything is implemented in C
C++.

Lexical analyser is described in a file scanner.l with regular expressions of lexems of VYPe2015 source language.

Syntax analyser is defined in a file parser.y with definitions of tokens and grammar rules with handlers – custom functions for every rule for generating intermediate code. We have implemented extension GLOBAL for possibility of global defined variables. The complete grammar in extended Backus-Naur form is in appendix on page 3 (rules for expression are skipped).

3 Compiler and generator

3.1 Symbol table

Symbol table contains all variables and functions with other child symbol tables (structure deque<symbol*> *table). This was the easiest way, how to deal with overlapping variable names in nested blocks of statements.

Because functions and variables are in the same structure, they are recognized by symbol's attribute type. Functions have their own data in attribute func and variables in var. Shared is only name attribute.

As was said functions and block have nested symbol tables and instructions list, which are used as thee-address code to generate final assembler.

3.2 Three-address code

In parser's handlers our compiler generate list of instruction from each rule. In this place are also generated labels for jumping in loops and functions. Every instruction is represented by child of Instruction class. There are instructions representing cast, assignment, arithmetical and logical operations between variables where the result is set to temporary variable (we do not optimize operations).

3.3 Generating assembly

From three-address code is generated final assembly code. because MIPS has not a stack, we have to implement our own stack simulation, to be able to call functions and block with their own scope variables. In the begging, there is a reserved part of memory used as stack. The generator then has to remember the addresses of each variable and put the address directly in assembled code. He has also simulate instruction and stack pointers, to know where to return back from nested blocks and functions.

For storing constants as chars, strings and integers we use .data directive because these data can not be modified in future.

3.4 Implementation

The compiler is implemented in C
C++ language.

4 Running the program

The program can be compiled by command `make`. It generates a binary file `vype` as required.

5 Division of work

Michal Danko	lexical analyser syntax analyser three-address code
Martin Šifra	code generator documentation

6 Summary

We have implemented the compiler for the programming language `VYPe2015`. For lexical and syntax analysis, we have used `flex` and `bison` tools. Compiler and generator are implemented in `C`
`C++`.

A Grammar

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

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>

}