VYPe: THE COMPILER PROJECT SPECIFICATION

Zbyněk Křivka
email: krivka@fit.vutbr.cz
September 26, 2016

1 General Information

Project Title: Compiler Implementation for VYPe16 Programming Language
Information Sources: VYPe forum in IS FIT, website of VYPe course
Project Website: `http://www.fit.vutbr.cz/study/courses/VYPe/public/project/`
Handover Deadline: Friday, December 30, 2016, before 23:59
Handover Medium: electronically into IS FIT (*Project (Compiler)* term)

Student Teams:

- The project is realized by a team with **two student members**.
- To enroll to the project, you have to register a variant in "Project (Compiler)" term in IS FIT. The registration consists of two steps. First, students create teams with two members in IS FIT. The leading student is called *Captain* and he/she confirms another team member. In the second step, the captain register chosen variant of the project. All variants are the same, you will just choose the number of the project. If there is an electronic communication between the team and a teacher, all emails are sent to all of them. If possible, do prefer to ask at the project forum in IS FIT. Then, state the question carefully and precisely.

Evaluation:

- Every member can get up to 20 points (17 for the compiler functionality and 3 for its documentation).
- If you will extend the project functionality (see Chapter 8.4), you can get 25 % higher score.
- The documentation can be evaluated at most by third of points for the compiler.
- Both, the points for the compiler and the documentation, can be redistributed adequately according the each student's effort.

2 Assignment

Study the specification of a simple programming language VYPe16, create and define its grammar, and build a compiler using syntax-directed translation. The compiler programmed by students will be an executable file or a bash script named `vype`. When executed

the compiler reads a given input file written in VYPe16 language and translates it into a target language. The target language is a symbolic address language for MIPS32-Lissom[1] processor (abbrev. MIPS). The translated target code will be simulated and evaluated using the given tools[2]. If the resulting program works without error, the program returns zero (`0`) returning value. If an error occurs, the returning value will be as follows:

- 1 = error during its lexical analysis (e.g. bad structure of the current lexeme).
- 2 = error during its syntactic analysis (e.g. bad program syntax).
- 3 = error during the semantic analysis and checks (e.g. missing variable declaration, type incompatibility, etc.).
- 4 = error during the target code generation (e.g. not enough statically allocated memory).
- 5 = internal compiler error such as errors not influenced by the input program (e.g. not enough memory for the translation structures, failure to open input/output files, etc.).

The filename with the input program in the language VYPe16 will be passed as the first argument in the command-line interface. The second argument (optional) determines the filename of the output target code (default is `out.asm`). Both arguments can be given by relative or absolute path. If the output file exists, it will be overwritten without any warning. All error messages will be printed on the standard error output; that is, the compiler will be a command-line application (with no graphical user interface).

Every regular expression used in the rest of the text is formulated according to GNU notation for extended regular expressions. Mainly, the spaces in the regular expressions are only for better readability and they are not parts of the regular expressions. As an exception, we sometimes need a space to separate two consecutive lexemes. The keywords are in bold. The identifiers and nonterminal symbols are written in italics. In the rest of the specification, *id* represents an arbitrary identifier. To increase readability, some lexemes are in quotas while the quotas are not part of VYPe16 language. For more information on the extended regular expressions, use command `info grep` at Merlin server (or any Linux-based operating system).

3 VYPe16Programming Language Specification

VYPe16 is inspired by and derived from language C.

3.1 General Properties and Data Types

VYPe16 is case-sensitive so uppercase/lowercase letters matter in identifiers and keywords. VYPe16 is a typed language so every variable has its data type determined by its variable declaration.

[1]The implementation of MIPS32 by Lissom research group.

[2]The tools including the assembler, linker, and instruction simulator are available at Project Website.

- *Identifier* is defined as non-empty sequence of letters (lowercase/uppercase), digits, and underscore character ("_") starting with a letter or underscore. Keywords have specific meaning and they cannot be used as identifiers. VYPe16 language has the following keywords: **`char`**, **`else`**, **`if`**, **`int`**, **`return`**, **`string`**, **`void`**, and **`while`**. In addition, the language includes several reserved words: **`break`**, **`continue`**, **`for`**, **`short`**, and **`unsigned`**. A multiple definition of a variable with the same name in the same scope is forbidden.
- *Data types* for literals defined further are denoted by **`int`**, **`char`** and **`string`**. Data types (denoted by *data_type*) are used in variable declarations and function definitions including definitions of returning values and function parameters. In addition, there is special type **`void`** (not included in *data_type* for the returning value of a function that represents no value. Hereafter, *data_type* and **`void`** are together denoted by *type*.
- *Integer literal* is a constant (decimal base) and defined by regular expression **`[0-9]+`**. It is represented by signed 32-bit **`int`**.
- *Character constant* consists of one *printable character* enclosed in apostrophes (`'`, ASCII value 39). *A printable characters* is a character with ASCII value greater than 31 (apart from 34 and 39) or an escape sequence. Available escape sequences follow: **`\n`**, **`\t`**, **`\\`**, **`\"`**, **`\'`**. The meanings of these escape sequences correspond to the same escape sequences in C language.
- *String literal* consists of *printable characters* (including escape sequences) and it is enclosed by double quotas (`"`, ASCII value 34). Internally, it is so called null-terminated string (a string is represented as an array of characters where the last non-inclusive character has ASCII value 0). The length of this kind of literal is almost unlimited since it is bounded only by the amount of free memory. The check whether the amount of memory is enough for allocated string can be omitted.
- VYPe16 language supports *line comments*. A line comment starts with two consecutive slashes ("**`//`**") and it includes every character until the end of line character.
- VYPe16 language supports *block comments* as well. These comments starts with two characters "**`/*`**" and ends with two characters "**`*/`**". An embedding of block comments is not supported.

4 Language Structure

VYPe16 is a structured programming language that supports definitions of variables, declarations and definitions of user functions including recursive calls. The start point of the whole program is a special function called **`main`**.

4.1 Basic Language Structure

A program consists of global declarations of user functions and of definition of user functions that cannot be embedded. Some commands of VYPe16 language are terminated by semicolon ("`;`", that is, a simple command terminator). Apart from comments we omit all white-spaces such as spaces, tabulators, and ends of lines. The white-spaces can occur between two lexemes and in the beginning and the end of the source code.

4.2 Function Declarations and Definitions

Every function has to be defined exactly once and declared at most once; that is, the function declaration is optional. The function declaration is useful when using mutual calls of two or more functions (e.g. mutual recursion). Considering lexical order in the source code, a definition or at least a declaration of a function has to be available before the first call-site (i.e. place from which the function is called; function call is made by a statement defined in Section 4.4). The function declaration and definition has the following form:

Function declaration:
type id **\(** *param_type_list* **\) ;**

The declaration of formal parameters *param_type_list* consists of a list of comma-separated data types.

Function definition:
type id **\(** *param_list* **\) {**
(*stmt* **)***
}

param_list is of the form:
(*data_type id* **(** **,** *data_type id* **)*** **)**

The parameters are passed by value. If the list of parameters is empty, use **void** type instead of the list.

The function body consists of a sequence of statements including the empty sequence and it is described by regular expression **(***stmt***)*** . For the description of syntax and semantics of statements, see Section 4.4.

>From the semantic point of view, the number of parameters and the order of data types in the header of a function definition has to match with the corresponding function declaration. If the is a function declaration, it has to lexically precede its definition. Every declared function has to be defined later in the source code. The overloading of functions is not supported except embedded function **print** and bonus extension OVERLOAD).

4.3 Program Main Function

Every program in VYPe16 has to contain exactly one main function named **main** with the type signature **int main \(void \)**, otherwise it a semantic error occurs. The structure of variable definitions and statements is described in the following sections.

4.4 Statements Syntax and Semantics

Every statement *stmt* has one of the following forms:

- *Definitions of local variables*:
  data_type id **(** **,** *id* **)*** **;**
  The scope of local variables is valid from this statement to the end of the containing block[3] where the local variable are defined. VYPe16 language supports overlapping

[3] *Block* is a sequence of statements enclosed in curly brackets of the form: **{ (** *stmt* **)* }**. The block determines so called *namespace*.

of variables in the embedded block when the variables have the same name. Then, such overlapped variable (from the outer block) is invisible in the inner block. By default, integer variables are initialized to zero 0, character variables to `'\0'`, and string variables to the empty string `""`. Is it not possible to define a variable of the same name as some already declared function or as some variable of the same name defined in the same block.

- *Assignment statement*:
  id **=** *exprz* **;**
  Semantics: The statement assigns a value of the right operand to the left operand. There is a condition that the left operand has to be already declared and it has to be of the same type as the right operand, namely **`int`**, **`char`**, or **`string`**, respectively. The left operand has to be always a variable (so called l-value). If the left operand is a **`string`** variable, then if necessary the compilation phase generates the target code for a memory reallocation (the size will be computed according to the size of the right operand) because a string variables are copied by value (not by reference).
- *Conditional statement*:
  `if \(` *expr* **`\)`**
  `{ (` $stmt_1$ **`)*`** **`}`**
  `else`
  `{ (` $stmt_2$ **`)*`** **`}`**
  Semantics: First, the value of *expr* is computed. If the result of the conditional expression is not **`int`** type, a semantic error occurs. Otherwise, if the result is `0` (zero), the conditional expression is interpreted as false. Any non-zero value of the result of the conditional expression is true. If the result is true, execute the first sequence of statements **`(`** $stmt_1$ **`)*`**; otherwise, execute the sequence **`(`** $stmt_2$ **`)*`**.
- *Iteration statement* **`while`**:
  `while \(` *expr* **`\)`**
  `{ (` *stmt* **`)* }`**
  Semantics: The rules for the truthfulness of the conditional expression *expr* are the same as above in the conditional statement. The iteration statement repeats the sequence of statements **`(`** *stmt* **`)*`** as long as the result of the conditional expression is evaluated to true.
- *Embedded/User function call*:
  *fun_id***`\( (`** *expr* **`( ,`** *expr* **`)* )? \) ;`**
  Semantics of the function call of *fun_id*: The function call statement is in fact the evaluation of the expression of the function call but the returned value is ignored if *fun_id* is non-void function. Before the function call, all real parameters, here denoted by *expr*, are evaluated from left to right. The parameters are passed by value into the function body and they are accessible using the names of formal parameters in the function definition. The number and types of real parameters has to match the function declaration/definition. If there are no formal parameters in the function definition, the parenthesis after the function identifier in the function call are empty. If a real parameter has unexpected type with respect to the function definition, a semantic error occurs. After the parameters evaluation and passing, the function body

Pr.	Operators	Assoc.	Meaning	Types
9	`()`	—	casting/function call	
8	`!`	unary	logical negation (prefix)	$N \to N$
7	`*` `/` `%`	$\to$	integer multiplicative	$N \times N \to N$
6	`+` `-`	$\to$	integer additive	$N \times N \to N$
5	`<` `<=` `>` `>=`	$\to$	relation	$T \times T \to \{0,1\}$
4	`==` `!=`	$\to$	comparison	$T \times T \to \{0,1\}$
1	`&&`	$\to$	logical AND	$N \times N \to \{0,1\}$
0	`||`	$\to$	logical OR	$N \times N \to \{0,1\}$

Tabulka 1: Operators of VYPe16 language

starts to be evaluated statement by statement. After the return from the function body (see **`return`** statement below), the computation continues immediately after the finished function call statement. If the function call is a statement (not an expression), its return value is dropped without any warning or error. The functions can be called recursively into an arbitrary level (limited just by the amount of free memory). For the semantics of embedded functions, see Section 4.6.

- *Return statement*:

  `return (` *expr* **`)? ;`**

  The statement ends the function execution of the currently called function (so called *callee*) and it returns a resulting value to the calling function (so called *caller*). Semantics: It evaluates expression *expr* (if not omitted in case of void function) to gain the returning value, and then, the execution of the function is finished immediately and if the function is not void function, then the returning value is available as the value of the function call at the caller site. If there is a type mismatch between the type of the finished function and the type of the returning value, a semantic error occurs. A void function is requests to end using just **`return`** statement without the expression parameter.

  In case of the return statement in **`main`** function written as "**`return`** *expr* **`;`**", the returning value is computed and the program exits (see **`BREAK`** instruction). If the function execution ends without execution of any **`return`** statement, then the function returns a default value depending on its type. More specifically, such integer, character, and string function returns `0`, `'\0'`, and `""`, respectively.

4.5 Expressions

An expression consists of integer, character, and string literals, variables, brackets, function calls, casting, and arithmetic, logic, and relation operators. The binary arithmetic and logic operators are left associative.

The complete list of operators with precedence/priorities (the highest at the first row) is in Table 1, where **Types** column uses the following abbreviations: N = **`int`**, T = *data_type*. These operators are described in more details in the rest of this section.

4.5.1 Arithmetic, logic, and relation operators

For operators `+`, `-`, and `*` it holds that for two operands of type `int`, the result is of type `int` as well.

Operator `/` denotes the integer division of two integer operands (the result is of type `int`). If we try to divide by zero, nothing will happen (see `DIV` instruction).

Operator `%` is used for gaining the integer remainder of the integer division (see `DIV` instruction). Obviously, for two operands of type `int`, the result is an `int` as well.

For operators `<`, `>`, `<=`, `>=`, `==`, and `!=` it holds that if both operands are of the same type and this type is `char`, `int`, or `string`, then the result is an `int`. The string comparison is lexicographical. The result of the comparison using relation operator is an integer value, `1` for truth; otherwise `0` for falsehood. The operators has the same semantics as in C language.

Logical operators `!`, `&&`, and `||` accepts integer operands of type `int`; that is, including the results of other logical, relation, and even arithmetical operations). The zero value represents false logical value and any other integer represents true logical value. The result of the application of some logical operator is either truth (integer value `1`), or falsehood (value `0`).

The other than the defined combination of types for the operators in expressions are considered to be an error.

4.5.2 Parentheses

The operator of explicit casting is written as `\(`*data_type*`\)` *expr*. It allows to convert `char` to `string` of length 1 and `char` to `int` (8-bit ASCII value of the given character (unsigned) extended to 32 bits). Further, `int` can be explicitly converted to `char` by the selection of the least important byte.

The result of the expression of function call is the returning value of such function.

The priority of operators can be explicitly changed by the use of parentheses in the subexpressions. Table 1 gives the priorities of operators (the first row describes the highest priority/precedence).

4.6 Embedded Functions

VYPe16 language supports several embedded functions given by the following function signatures. These functions are implemented directly in the compiler of the language and can be used without any previous definition or declaration.

- `void print\(` *data_type* `( , ` *data_type* `)* \)` – The function prints one by one all its parameters to the standard output using the special instructions for the output. The number of parameters is arbitrary, but at least one, and the parameters are of arbitrary supported types.
- `char read_char\( void \)` – It reads the input character from the user ended by the end of line (`EOL`[4]) using the special read instruction.

[4]The simulator of the instruction set MIPS32-Lissom considers `EOL` to be ⟨*LF*⟩ in Unix-like systems and ⟨*CR*⟩⟨*LF*⟩ in Windows operating systems.

- **`int read_int\( void \)`** – It reads the input from the user ended by the end of line (**`EOL`**) using the special instruction to read an integer of **`int`** scale without the overflow check.
- **`string read_string\( void \)`** – It reads the input from the user ended by the end of line (**`EOL`**) using special instruction to read a string (omitting the final **`EOL`** or **`'\n'`**).
- **`char get_at\(string , int \)`** – The function returns a character from a given position of a string. The string is given by the first parameter, the position is given by the second parameter (zero-based index). There is no out-of-range check, so the behavior for negative or too great index is undefined.
- **`string set_at\(string , int , char \)`** – The function returns a string that is created/modified from the string given by the first parameter by replacement of the character on the position given by the second parameter (zero-based index) by a new character given in the third parameter. As in the previous function, there are no out-of-range checks.
- **`string strcat\(string , string \)`** – It returns a string that is created as the concatenation of the first parameter and the second parameter.

4.7 Implementation Notes

The design and the implementation of the compiler is completely up to you. Since extensive amount of work on the compiler and considering the modern trends in modern compiler design, we recommend to use as much automatic and generation tools supporting the creation of the compiler as possible. More specifically, consider to use tools such as **`flex`** (new implementation of the classical **`lex`**) to create scanner and **`bison`** (new implementation of the classical **`yacc`**) to create parser including semantic analysis based on the bottom-up approach (LALR method). If you prefer to use Java programming language, consider using complex generation tool such as ANTLR (Java libraries of versions 3.5.2 and 4.5.1 can be accessed directly on Merlin server in directory `/pub/courses/vyp/`). If you would like to use other tools, contact the project consultant for the approval.

The support for the input-output embedded functions is already supported by special instructions in the target MIPS language and integrated into corresponding tools available at Project Website. You just have to use these instructions during the code generation properly.

5 Examples

This chapter contains three small and simple examples of programs in VYPe16.

5.1 Iterative Factorial Computation

```
/* Program 1: Iterative Factorial Computation */

int main(void) { // Program Main function
  int a, res;
```

```
  print("Enter an integer to compute its factorial:");
  a = read_int();
  if (a < 0) {
     print("\nFactorial of a negative integer is undefined!\n"); }
  else
  {
     res = 1;
     while (a > 0)
     {
        res = res * a;
        a = a - 1;
     } // endwhile
     print("\nThe result is: ", res, "\n");
  } // endif
} // main
```

5.2 Recursive Factorial Computation

```
/* Program 2: Recursive Factorial Computation */

int factorial (int);
int main(void)
{
  int a; int res;
  print("Enter an integer to compute its factorial:");
  a = read_int();
  if (a < 0)
  {
     print("\nFactorial of a negative integer is undefined!\n");
  }
  else
  {
     print("\nThe result is: ", factorial(a), "\n");
  }
}
int factorial (int n)
{
  int decremented_n, temp_result;
  if (n < 2)
  {
     return 1;
  }
  else
  {
     decremented_n = n - 1;
     temp_result = factorial(decremented_n);
  }
  return (int)n * temp_result;
} // end of factorial
```

5.3 Program using Strings and Embedded Functions

```
/* Program 3: Program using Strings and Embedded Functions */

int main(void) {
```

```
  string str1, str2;
  int p; char zero;

  str1 = "This␣is␣some␣text";
  str2 = strcat(str1, "␣that␣can␣be␣a␣little␣longer.");
  print(str1, '\n', str2, "\n");

  str1 = read_string();
  while ((int)(get_at(str1, p)) != 0)
  {
    p = p + 1;
  }
  print("\nThe␣length␣of␣\"", str1, "\",␣is␣", p, "␣characters.\n");
}
```

6 Target Symbolic Address Language of MIPS32 Processor

We will briefly describe the symbolic address language of the Lissom implementation of MIPS32 (Release 1) processor, abbreviated as MIPS language. The goal of this chapter is not to give exhaustive description of the language, as there are detailed manuals and documentations where detailed information can be found (see the reference section located on the VYPe Project Website), but rather to give an overview information for quick introduction into the problematic.

As in other symbolic address languages, there are many different instructions, identifiers of registers, constants, directives and labels. The letter case is irrelevant for instructions and identifiers (i.e. *case insensitive*), however this does not apply to labels, where the letter case matters (i.e. *case sensitive*). Each instruction occupies exactly one line. The same applies to directives and labels. Single line and block comments can be written similarly as in C language.

6.1 Basic Characteristics of MIPS32-Lissom Processor

Target MIPS32-Lissom processor contains[5]

- 32 universal 32-bits registers (**`$0`** to **`$31`**)
- architecture registers `rhi` and `rlo` for division and multiplication (unreachable from MIPS language)
- program counter (register `PC`) (again unreachable from MIPS language)
- 1 MB of memory is organized by bytes (1 byte = 8 bits), the smallest addressable unit is 1 byte, larger structures are stored in the memory using the *big endian* style
- the memory is shared by the program and the data (Von Neumann architecture)
- data stack is not present and has to be simulated
- floating point operations are not present

[5]The detailed information regarding the model of this processor can be acquired from the released description in ISAC language.

6.2 Target File Format

The file that serves for the compilation of MIPS language into a simulated program (using the assembler tool and the linker) is divided into several sections. Each section starts with a special directive.

6.2.1 Directives

Directives of MIPS language allow the source code to be split into multiple sections (**`.text`** for the code and **`.data`** for the data). Directive **`.org`** *number* can be used to set absolute position of the following entry in the memory (at most one such directive is allowed in each section). These and other directives like **`.ascii`**, **`.asciz`**, **`.byte`** and **`.int`** are described in `http://sourceware.org/binutils/docs-2.20/as/Pseudo-Ops.html`. List of allowed directives for MIPS language is documented in Lissom Assembler Directives Reference (see Project Website).

An example of the structure of MIPS language file:

```
.text
.org 0
  MOVI $SP, 0x4000
jaL main
BREAK
main:
// ... save $RA ...
/* ... compute main body
   ... compute return value into $2 ... */
// ... load $RA ...
jr $RA
.ascii "hello\000"
.asciz "world"
.int 1
.int 42
```

6.3 A Brief Description of the Instruction Set

The detailed description of the instructions and their semantics are found on Project Website, in the "MIPS Technologies, Inc.: MIPS32™ Architecture For Programmers Volume II: The MIPS32™ Instruction Set, 2003 (Rev. 2.0)" manual. Consider only the instructions implemented in the version 1 of the processor (see pages 24 to 27, tables 3-1 to 3-5, 3-7, 3-8 and instructions **`BREAK`** from the table 3-9). Their specific implementation in MIPS32-Lissom may be checked, using the ISAC language processor model (see Project Website).

Some of the irrelevant instructions such as the floating point operations, interrupts and traps, co-processor extensions, and system calls (including a memory allocation which has to be handled by the program) are not implemented. Because of the missing traps, MIPS32-Lissom simulator is unable to detect run-time errors and the most of those errors are skipped without any warning (e.g. division by zero is simply skipped).

6.3.1 Special Input-Output Instructions

The description of semantics and syntax of special instructions, where *r*, *rd* and *rl* represent the universal registers, follows.

- *stdin* $\rightarrow$ *r* reads a character and stores it in *r*.
  `READ_CHAR` $*r*
- *stdin* $\rightarrow$ *r* reads a value of a number and stores it in *r*.
  `READ_INT` $*r*
- *stdin* $\rightarrow$ *rd*, *rl* reads a string and stores it in the memory given by the address stored in *rd* register and stores the length of the read string without the ending \0 sign in *rl* register.
  `READ_STRING` $*rd* **`,`** $*rl*
- *r* $\rightarrow$ *stdout* prints the value of *r* register as a character.
  `PRINT_CHAR` $*r*
- *r* $\rightarrow$ *stdout* prints the value of *r* register as a number.
  `PRINT_INT` $*r*
- *r* $\rightarrow$ *stdout* prints the string given by the address stored in *r* register.
  `PRINT_STRING` $*r*

Every read value (**`char`**/**`int`**/**`string`**) is on its own line ended by **`EOL`**. The conversion of numbers is based on **`atoi`** function of the C language. The longest valid sequence of characters is read and all remaining symbols on the line are ignored. If the input does not contain any valid symbol on the start of the line (i.e. it is invalid), the result is the zero value (**`0`**) or the empty string. Possible overflowing is not taken care of (see the model in ISAC language).

6.3.2 Program Termination

The return value of **`main`** function shall be stored in $2 register and the program shall be terminated by a non-parametric instruction **`BREAK`**. The simulator in a text mode will print the values of all of the registers and the position of the program counter.

6.3.3 Typical Use of the Register for MIPS ABI

Even though the universal registers may be used arbitrarily, there is a certain standard which improves the readability by naming some of the registers. Table 2 shows the most important denomination of the registers that we recommend you to follow in your projects. There is a more detailed description of MIPS ABI on Project Website for those, who are more interested in the problematic.

6.4 Supplied Tools

For the simulation of the code generated by your program in MIPS language, first, it is necessary to build (assembler) and link (linker) the code. We shall describe the basic control of the tools from the command line.

Register	Alias	Meaning or reservation
$0		always contains zero (zero register)
$1	$at	reserved for MIPS language
$28	$gp	global pointer (in case of a global data)
$29	$sp	points to the top of the stack (stack pointer)
$30	$fp	points to the context (frame pointer)
$31	$ra	the most actual return value (return address)

Tabulka 2: Reserved registers of MIPS32 instruction set

6.4.1 Assembler and Linker

Suppose that your output program in MIPS language is in `out.asm` file. You can build the program into an object file using the command:

```
assembler2 -i out.asm -o out.obj
```

After that you have to create a simulated file using the command:

```
linker out.obj -o out.xexe
```

6.4.2 Simulator

Because the students have no access to a physical MIPS32 processor, the output code is only simulated by the supplied simulator. Simulator starts the simulation on a memory address `0`. In a case that your program starts on a different address, appropriate jump instruction has to be added. The simulation itself, with the input in a file `input.in`, can be started from the command line using the command:

```
intersim2 -i out.xexe -x report.xml -n mips < input.in
```

When the simulation is finished a table with the values of the universal registers is printed together with the value of the program counter and some other statistics.

7 Project Submission Instructions

Please, do not underestimate the following information. Since the projects will be mainly evaluated by a script, the ignoring some of these instructions may lead to the inability of the script to properly evaluate the project.

7.1 General Information

The project is handed in only by the team leader. All handed files will be compressed using the TAR+GZIP or ZIP programs into a single archive, named `login.tgz`, or `login.zip`. All files contained in the archive must be named according to the regular expression **`[A-Za-z_.0-9]+`**.

The whole project must be handed in a given date (see above). Failure to do so will lead to project being considered not handed in or at least final score penalization proportional to the length of the delay is to be expected. Similarly, any form of plagiarism will lead to project being scored by zero points and further disciplinary actions will be considered.

7.2 Points Division

The submitted archive must contain file named **`division`** where the points division among the team members will be taken into account (even when the points are divided evenly). Each line of the file must contain the login of the team member followed (without any spaces) by a colon followed (again without any spaces) by requested integer value of percentage of points without the % sign. Each line (including the last one) must immediately end with a single ⟨LF⟩ (ASCII value 10, i.e. Unix line end). For instance, the file may look like this:

```
xnovak01:60<LF>
xnovak02:40<LF>
```

The sum of all percentage must be equal to 100. If the total sum of percentage is incorrect, the points will be split evenly. The format of the file must be correct and must contain all members of the team, including the members with 0 %.

8 Implementation Requirements

In addition to the implementation and documentation requirements, this chapter contains a list of extensions for extra points and few tips regarding the successful implementation of the project.

8.1 Mandatory Methods of Compiler Implementation

We recommend using some of the existing generators (Flex, Bison, ANTLR, ...) for lexical and syntactic analyzers implementation. Usage of any existing compiler *front-end* or *back-end* like LLVM, GCC, SDCC and others is forbidden. The project will be implemented using **any programming language** (e.g. C, C++, Java, Python[6]) that can be compiled and run on the server `Merlin` (without any additional setup) which will be considered as a reference server for this project. The design of the implementation is completely up to the teams. A required part of the solution is a `Makefile`, where compilation parameters may be set (see `info gmake`). If `Makefile` is not present, or it is not possible to compile the target program with it, the project will not be evaluated any further! The compilation or initial setup will be run using the `make` command. Then, the program may be run using the binary file or script named `vype`, which has one mandatory and one optional argument, as is described in Chapter 2.

8.2 Text Documentation

The part of the solution is a documentation in PDF format, contained in a single file named **`documentation.pdf`**. The documentation in any other format will be ignored and no points will be awarded for the documentation part of the project. The documentation will be written in English. The documentation should be 4-7 A4 pages long. **The documentation shall** compulsorily contain:

[6] Other languages should be approved first by the project consultant.

- 1st page: names, surnames and logins of authors + points division information and list of identifiers of implemented extensions.
- Description of the front-end of the compiler including the used grammar and methods and tools used.
- Description of your implementation of the back-end - design, implementation, intermediate code, symbol table, special techniques used, optimizations, algorithms.
- Work division among the team members (describe who and in a what way worked on specific parts of the project).
- In the case of nonstandard implementation, state how to run the project on Merlin server or other environment explicitly allowed by the project consultant.
- Literature used, including citations of any non-original materials (pictures, statistics etc.).

The documentation shall not:

- Contain copy of any part of the project specification or any text, pictures[7] or diagrams that you are not the authors of (copy from lectures, network, WWW, ...).
- Be based on an enumeration and general description of each method (it is your own implementation; you should therefore describe your own approach to the solution; obstacles you encountered; problems you had to solve and how you solved them; etc.)

The state of the source codes, like its readability, clarity and sufficient, but not excessive number of comments will be also considered as a part of the documentation evaluation.

Each source code must contain commented out name of the project and names and logins of authors as its header.

Every error messages that arose from the run of the program must be outputted to standard error output. The program shall not output any characters or text during its execution, except the texts prescribed by the control program written in VYPe16 language. Basic tests will be automatically performed using the scripts that will gradually run a set of test programs (compilation of the program with your compiler, build and simulation using the given tools) and compare your output with the expected output. For the output comparison the `diff` program is used (see `info diff`). Therefore, any unexpected character outputted by the program compiled by your compiler will lead to failure of the given test and a point loss.

8.3 Recommendations Regarding the Work on the Projects

Theoretical knowledge needed for successful project implementation will be gained during the semester at VYPe lectures and forum. It is important that the whole team collaborates on the project. It is recommended that the team agrees on having a regular meetings and communication channels that will be used during the project implementation (instant messaging, conference tools, version control systems etc.).

[7] Exception being the faculty logo on the front-page of the documentation.

Situation, when some of the team members ignore the project, may be resolved by file `division` or by personal consultation with the project consultant. We strongly recommend to check the real work progress during regular team meetings, so you can eventually redistribute the work among the team members. **Maximal number of points** that can be obtained by each person (including the extra points for extensions) is **25**.

Do not leave the work on the project until the last week. The project consists of several parts (e.g. lexical analysis, syntactic analysis, semantic analysis, intermediate representation, symbol table, code generation, documentation, testing!) and it is created in a such way that some parts of the project can be designed and implemented during the semester, based on your previous knowledge and knowledge obtained during VYPe lectures, VYPe forum, Project Website and self study.

8.4 Registered Extensions

In case when some of the registered extensions are implemented, the submitted archive must contain file called **`extensions`** which shall list the identifiers of the extensions that you implemented. Each line shall contain one identifier, followed by a UNIX style new line character (i.e. ⟨LF⟩ sign).

During the semester the list of registered extensions and their identifiers can be gradually expanded. See VYPe forum, where new the extensions and their point values will be listed. You may send your ideas for new extensions that you would like to implement on the forum. Then, the project consultant will decide whether the proposed extensions will or will not be accepted. If accepted, the point value and new identifier of the new extension will be determined based on its complexity. The points for the extensions are part of the score for the project so they are part of the 25 points limit.

8.4.1 List of the Extensions that Can Be Rewarded by Extra Points

Following descriptions of the extensions of VYPe16 language always start with a respective identifier and end with their maximal point value.

- SHORTEVAL: Support the evaluation of conditional expressions using the *short evaluation* (+1.5 points).
- GLOBAL: Support of global variable declaration (+1.0 point).
- MINUS: The compiler will support a prefix unary minus and plus operators (priority 8). Describe in your documentation how did you deal with this problem (+1.0 point).
- FOR: The compiler will support **`for`** cycle. In addition, support keyword **`break`** and **`continue`** with the same semantics as in C language (+1.0 point).
- HEAP: Implementation of the missing heap memory management including both automatic allocation and deallocation. Describe your approach in the documentation (+3.0 points).
- IFONLY: Simplified conditional statement **`if`** without the **`else`** part. In addition, instead of a whole block, single statement may appear inside the conditional statement or cycles. Describe your approach in the documentation[8] (+1.5 points).

[8] For example, how did you solve the nested **`if`** and **`if-else`** pairs? Modern languages pair the **`else`** with the closest **`if`**.

- INITVAR: Initialization of a variable with an expression during its definition (+0.5 points).
- OVERLOAD: Overloading of user defined functions (+2.0 points).
- UNSIGNED: Support of the unsigned types **`unsigned int`** (and in case of SHORT extensions also **`unsigned short`**) including their conversions (without overflow checks) (+1.0 point)
- BITOP: Support of bit operations **`~`**, **`&`** and **`|`** inside the expressions (+0.5 points).
- SHORT: Support of a smaller integer-number type **`short`** (16-bit integer). Explicit conversion of **`short`** to **`int`** is done by sign extensions of 16-bit number to 32-bit number. As a part of implicit conversions, it is possible to use a value of **`short`** type wherever the **`int`** is expected (this is achieved by a sign extension). You may use special instruction **`READ_SHORT`** $*r* that reads the value of a number, truncates it to 16 bits and stores it in the *r* register. Further, instruction **`PRINT_SHORT`** $*r* prints the 16-bit value in the register *r* as a number (+1.0 point).
- ...