VYPa: THE COMPILER PROJECT SPECIFICATION

Zbyněk Křivka, Radim Kocman, Dominika Regéciová
email: krivka@fit.vut.cz
September 15, 2025

1 General Information

Project Title: Compiler Implementation for VYPlanguage Programming Language
Information Sources: forum in VYPa Moodle, website of VYPa course
Project Website: `https://www.fit.vut.cz/study/course/VYPa/public/project/`
Hand-in Deadline: Monday, December 15, 2025, before 23:59
Hand-in Medium: electronically into StudIS (*Compiler Project* assignment)

Student Teams:

- The project is realized by a team of **two**.
- To enroll in the project, you must register a variant in "Compiler Project" in StudIS. To do so, the team leader will first create a team of two in IS VUT (Student module). The name of the team is inferred from the leader's login. The team leader is responsible for confirming/registering the other team member and registering the only variant of the project. If possible, do prefer to ask any project-related question in the project forum in Moodle. State the question carefully and precisely. If communicating with the teacher through emails, always include the other team member in the conversation.

Evaluation:

- Every member can get up to 20 points (17 for the compiler functionality and 3 for its documentation).
- If you extend the project functionality (see Section 8.4), you can get an up to 25 % higher score.
- The points for the documentation will never exceed the third of points gained for the compiler functionality.
- The points for the Compiler project can be redistributed based on the each student's individual effort.

2 Assignment

Study the specification of a simple programming language VYPlanguage with the support of the object-oriented paradigm including objects, classes, polymorphism, and

simple inheritance. Based on your findings, create and define its grammar and build a compiler using the syntax-directed translation. The created compiler will be an executable file or a bash script named `vypcomp`. During its execution, the compiler reads a given input file written in the source language called VYPlanguage and translates it into the target language called VYPcode (see Chapter 6). VYPcode target code can be processed by the interpreter called `vypint` which is available at Project Website or at Merlin server in `\pub\courses\vyp\vypint` directory. `vypint` will be used together with a `diff` tool for the evaluation of your compiler project. If the compilation from VYPlanguage into VYPcode was successful (i.e. without an error), the compiler returns zero (`0`) as its returning value. If an error occurs, the returning value will be as follows:

- 11 = error during its lexical analysis (e.g. bad structure of the current lexeme).
- 12 = error during its syntactic analysis (e.g. bad program syntax).
- 13 = error during the semantic checks of the type incompatibility.
- 14 = error during the rest of semantic checks (e.g. missing definition, multiple definition, etc.).
- 15 = error during the target code generation (e.g. not enough statically allocated memory).
- 19 = 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 VYPlanguage program is 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.vc`). Both arguments can be given by relative or absolute paths. 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 upcoming text is formulated according to GNU notation for extended regular expressions. The spaces in the regular expressions are due to better readability only and they are not part of the regular expressions. As an exception, space occasionally must be added to separate two consecutive lexemes. The keywords are in bold, the identifiers and nonterminal symbols are written in italics, and, in the rest of the specification, *id* represents an arbitrary identifier. To increase readability, some lexemes are in apostrophes even though the apostrophes are not part of VYPlanguage. For more information on the extended regular expressions, use command `info grep` on Merlin server (or any Linux-based operating system).

3 VYPlanguage Programming Language Specification

VYPlanguage is inspired by and derived from languages C and Java.

3.1 General Properties and Data Types

VYPlanguage is a case-sensitive language, so uppercase/lowercase letters matter when used in identifiers and keywords. VYPlanguage is also a typed language, therefore,

every variable has to have its data type determined by its variable definition.

- *The identifier* is defined as a non-empty sequence of letters (lowercase/uppercase), digits, and underscore character ("_") starting with a letter or an underscore. However, keywords have specific meaning and they cannot be used as identifiers. VYPlanguage has the following keywords:

  `class`, **`else`**, **`if`**, **`int`**, **`new`**, **`return`**, **`string`**, **`super`**, **`this`**, **`void`**, **`while`**.

  In addition, in bonus extensions there can be some additional definitions of keywords.
  Multiple definitions of variable with the same name in the same scope are forbidden.
- *Data types* for literals defined further are denoted by **`int`** and **`string`** (so called *primitive data types*, denoted by *prim_type*). Further, every identifier of a defined class (even later in the source code) can be used as a data type (so called *object type*). Data types (denoted by *data_type*) are used in (instance) variable definitions, function/-method definitions including the definitions of returning values and function/method parameters. In addition, there is special type **`void`** (not included in *data_type*) that represents no value and it is used to define that a function/method has no returning value or no parameters. Hereafter, *data_type* and **`void`** are together denoted by *type*.
- *An integer literal* is a constant (decimal base) and defined by regular expression **`[0-9]+`** but without superfluous zeros at the beginning. The representation coincides to the interpreter integer representation (64-bit, signed).
- *String literal* consists of *printable UTF-8 characters* (including escape sequences) that are enclosed by double quotes (", UTF-8/ASCII value 34). *A printable character* is a character with UTF-8/ASCII value greater than 31 (apart from 34) or an escape sequence. Available escape sequences[1] are as follows: **`\n`**, **`\t`**, **`\\`**, **`\"`**. UTF-8 characters will not be tested but are supported by VYPcode and vypint. In VYPlanguage source code, each UTF-8 character can be written as a hexadecimal escape sequence "**`\x`*`hhhhhh`***" where *hhhhhh* is a hexadecimal number of exactly 6 hexadecimal digits. Internally, we recommend to use chunks for the representation in VYPcode. The length of this kind of literal is almost unlimited since it is bounded only by the amount of free memory for chunks. The check whether the amount of memory is enough for the string can be omitted.
- VYPlanguage supports *line comments*. A line comment starts with two consecutive slashes ("**`//`**") and it ends by the end of the line character.
- VYPlanguage supports *block comments* as well. These comments start with two characters "**`/*`**" and ends with two characters "**`*/`**". An embedding of block comments is not supported.

4 Language Structure

VYPlanguage is a structured programming language that supports definitions of variables, user functions and classes including (mutually) recursive calls (even without for-

[1] The meanings of these escape sequences correspond to the same escape sequences in C language.

ward declarations of functions/classes/methods). The starting point of the entire program is a special function called **`main`**.

4.1 Basic Language Structure

A program consists of global definitions of user functions and user classes. Both, functions and classes cannot be embedded. Some commands of VYPlanguage 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 Definitions

Every function has to be defined exactly once. Considering lexical order, a definition of a function can be placed later in the source code than the first call-site (i.e. place from which the function is called; function call is made by a statement defined in Section 4.6). The function definition has the following form:

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

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

The parameters of primitive types are passed by value and parameters of object types are passed by reference. If the list of parameters is empty, use **`void`** type instead of the empty 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 4.6.

The overloading of functions/methods is not supported except for embedded function **`print`** and bonus extension OVERLOAD.

4.3 Program Main Function

Every program in VYPlanguage has to contain exactly one main function named **`main`** with the type signature **`void main \( void \)`**, otherwise a semantic error occurs. The structure of variable definitions and statements is described in Section 4.6.

4.4 Object Model

There are complex user types called *classes*. Each class is named by its identifier. We can use simple inheritance to create a new class based on some already defined class. The structure based on a class created during run-time is called *object* or *instance*. A class defines the set of instance variables to store data in an instance and the set of methods to provide appropriate manipulators to these data. All methods and instance variables (both

called as *members*) are defined as *public* which means there is no restriction for the access to these members despite the standard visibility/scope rules. A method can be redefined (so called *overridden*) with a new method body and with the same type signature in any descendant class. A redefinition of instance variables is not allowed. All methods are polymorphic which means that the polymorphism concept is applied during every method call/invocation in VYPlanguage. An object is created by **`new`** operator. A method of the same name as the class returning **`void`** and without parameters is called *constructor*. There is one embedded class called **`Object`** as the root of the inheritance hierarchy.

During the execution of a method body, the context object is available through **`this`** pseudo-variable. If we need to access the context object in the parent mode, we use **`super`** pseudo-variable that allows to access original methods from the ancestor classes (if redefined in the class of the context object).

Subsumption rule: When an instance of class *A* is expected by the type system, the user can provide not only any instance of *A* but any instance of any subclass (even indirect) of *A* as well.

If *B* is an (indirect) subclass of *A*, the instance of *B* is *type compatible* with data type *A* since all members available in instances of *A* are available in all instances of all subclasses of *A* as well.

4.5 Class Definitions

The identifier of the defined class is visible everywhere in the source code.

Class definition:

`class` *id* **`:`** *superior* **`{`**
  `(` *defs* **`)*`**
`}`

where *defs* represents instance variable definitions and method definitions. Syntactically, the instance variable definition looks exactly like a local variable definition (see Section 4.6) placed inside the class definition. The valid scope of instance variables (using **`this`**) is from their definition to the end of the class definition. Since instance variables are public, they are accessible through the object variable, the dot operator, and *id* even outside their scope. The default instance variables initialization coincides with the initialization of local variables described in Section 4.6. It is not allowed to define a variable of the same name as some defined method in the same class.

The method definition coincides with the syntax of a function definition but inside the class. For an example, see Section 5.4.

One of the methods can be parameter-less constructor:

`void` *id***`\(`** **`void`** **`\)`** **`{`** ... **`}`**

The constructor is called automatically every time an instance of the class *id* is created using **`new`** operator. In addition, the constructor call is preceded by the automatic call of the constructor of the direct ancestor class. The constructor usually initialize instance variables and prepare new instance for use. If some constructors are missing, we try to call at least all available constructors in the constructor chain. The *constructor chain* starts with the constructor of the most origin ancestor class and continues with its (direct if possible) descendants until we reach the constructor of class *id* that is the data type of the created object.

In a method body, every method invocation or instance variable access needs to use **this** or **super** before the dot operator. On the other hand, after **this** or **super** and the dot, there can be only an instance variable id or a method id; that is, you cannot cascade **this** or **super**.

4.6 Statements Syntax and Semantics

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

- *Definitions of local variables*:
  data_type id **(, *id*)* ;**
  Semantics: The valid scope of local variables is from this statement to the end of the block[2] where the local variables are defined. VYPlanguage supports overlapping of variables in the embedded block when the variables have the same name. Then, such an overlapped variable (from the outer block) is invisible in the inner block. By default, integer variables are initialized to zero 0, string variables to the empty string **""**, and object variables to 0-reference as undefined chunk id. It is not possible to define a variable of the same name as some defined function/class or variable defined in the same block.
- *Assignment statement*:
  id **=** *expr* **;**
  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 defined and it has to be of the same or compatible type as the right operand, namely **int**, **string** or class. The left operand has to always be a variable (so-called L-value). Note that if the left operand is a **string** variable, then the right operand is 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, *r*, is of type **int**, then any non-zero value of *r* represents true and 0 (zero) means false. Further, if *r* is an object and the representing chunk is undefined (chunk id 0), *r* represents false, otherwise *r* represents true. Other types of *r* lead to a semantic error. If *r* represents true, execute the first sequence of statements **(** *stmt*$_1$ **)***; otherwise, execute the sequence **(** *stmt*$_2$ **)***.
- *Iteration statement* **while**:
  while \(*expr* \)
  { (*stmt*)* }
  Semantics: The expression *expr* is computed the same way as in the previous con-

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

ditional statement. The iteration statement repeats the sequence of statements **(** *stmt* **)*** as long as the result of the conditional expression is evaluated as true.

- *Embedded/User function call*:

  *fun_id***\((** *expr* **(, ** *expr* **)*)? \) ;**

  The semantics of the function call of *fun_id*: The function call statement is, in fact, the evaluation of the expression passed to 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 object parameters are passed by reference and primitive 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 have to match the ones of the function definition. If there are no formal parameters in the function definition, the parenthesis 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 are evaluated and passed to the function, the function body proceeds the evaluation statement by statement. After the return from the function body (see **return** statement below), the computation continues immediately after the finished function call statement. The functions can be called recursively at an arbitrary level (limited only by the amount of free memory). For the semantics of embedded functions, see Section 4.8. Note that when the function call is a part of an expression, the return value is not dropped (see Section 4.7).

- *Method invocation/method call*:

  co_expr **.** *met_id***\((** *expr* **(, ** *expr* **)*)? \) ;**

  The semantics of the method invocation of *met_id*: First, *co_expr* is evaluated as a *context object* of this method invocation. It is a semantic error whether the result of *co_expr* is not an object or an object of the class that allows invocation of *met_id*. The evaluation of real parameters and returning value is the same as for a function call.

- *Return statement*:

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

  The statement ends the function/method execution of the currently called function/-method (so-called *callee*) and it returns a resulting value to the calling function/-method (so-called *caller*). Semantics: It evaluates expression *expr* (if not omitted in case of void function) to gain the returning value, the execution of the callee is finished immediately afterward. If the callee is not void, the returning value is available as the value of the function/method call at the caller side. If there is a type mismatch/incompatibility between the defined type of the callee and the type of the returning value, a semantic error occurs. A void function/method can be exited using just **return** statement without the expression parameter. The **main** function is void and **return;** just exits the program. If the function/method execution ends without execution of any **return** statement, then it returns a default value depending on its type. More specifically, such integer and string function/methods returns `0` and **""**, respectively. If it should return an object, the default return value is zero chunk id.

Pr.	Operators	Assoc.	Meaning	Types
11	`new`	unary	object creation (prefix)	
10	`.`	left	method call/instance variable access	
9	`()`	—	casting/function call	
8	`!`	unary	logical negation (prefix)	$N \to N$
7	`*` `/`	left	integer multiplicative	$N \times N \to N$
6	`+`	left	integer addition, concatenation	$P \times P \to P$
6	`-`	left	integer subtraction	$N \times N \to N$
5	`<` `<=` `>` `>=`	left	relation	$P \times P \to \{0,1\}$
4	`==` `!=`	left	comparison	$T \times T \to \{0,1\}$
1	`&&`	left	logical AND	$NC \times NC \to \{0,1\}$
0	`||`	left	logical OR	$NC \times NC \to \{0,1\}$

Tabulka 1: Operators of VYPlanguage

4.7 Expressions

An expression consists of integer and string literals, local variables, instance variables[3], brackets, function/method[4] calls, object creation, context object[5], 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*, P = *prim_type*, and NC = `int` or object type. These operators are described in more details in the remainder of this section.

4.7.1 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`). In addition, `+` can be used to concatenate two strings.

For operators `<`, `>`, `<=`, `>=`, `==`, and `!=` it holds that if both operands are of the same type `int` or `string`, then the result is an `int`. The result of the comparison using relation operator is an integer value, `1` for truth; otherwise `0` for falsehood. In addition, `==` and `!=` can compare objects for identity; that is, we get `1` for `==` if and only if both operands refer to the same object (i.e. the same chunk id). The string comparison is lexicographical.

Logical operators `!`, `&&`, and `||` accepts `int` operands and object operands; that is, including the results of other logical, relation, and even arithmetical and object operations. The zero integer value represents false logical value, and any other integer represents true

[3]In a method body, prefix "`this.`" can be used. Outside a method body, use only the object variable followed by the dot and the name of the instance variable.

[4]To call a method in a method body, prefixes "`this.`" or "`super.`" can be used. Outside a method body, use only the object variable followed by the dot and the name of the method.

[5]In a method body, we can use `this` as the reference to the object itself as a real parameter or an operand of a return statement.

logical value. An undefined object stands for `0`, others for `1`. The result after the application of some logical operator is either truth (integer value `1`), or falsehood (value `0`).

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

4.7.2 Object Creation

The object is created based on the given class, and the new instance is the results of such an expression. Syntactically,

`new` *id*

where *id* is a class name. During the creation, the appropriate memory is allocated on the heap of the appropriate size to store all instance variables and methods (including inherited variables, virtual method table if necessary, etc.), then the parameter-less constructor chain of *class_id* is called.

4.7.3 Parentheses

The parentheses have three meanings depending on the context of their use.

(1) The operator of explicit casting is written as **`\(`***data_type***`\)`***expr*. It allows converting **`int`** to **`string`**. Next, if *data_type* is object type, the conversion is dynamic cast of the operand to the compatible subclass so the run-time check of the object type is needed; if the real type of the object is incompatible with the *data_type*, cause run-time error 28 during the interpretation.

(2) The result of the expression of function/method call is the returning value of such function/method.

(3) 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).

4.8 Embedded Functions

VYPlanguage supports several embedded functions given by the following function signatures. These functions are implemented directly in the compiler and they can be used without any previous definition.

- **`void print \(`** *prim_type* **`( , `** *prim_type* **`)* \)`** – The function prints all its parameters one by one to the standard output (`stdout`) in UTF-8 encoding using instructions `WRITES`/`WRITEI` for the output. The number of parameters is arbitrary, but at least one has to be specified. The parameters are of arbitrary supported primitive data types.
- **`int readInt \( void \)`** – Reads the input from the standard input (`stdin`) ended by the end of line (**`EOL`**) using instruction `READI`.
- **`string readString\( void \)`** – Reads the input in UTF-8 encoding from the standard input (`stdin`) ended by **`EOL`** using instruction `READS` (omitting the final **`EOL`**).

- **int length \(*s* \)** – Returns the length (the count of characters) of the string given by parameter *s*. For instance, **length("x\nz")** returns 3.
- **string subStr \(** *s*, *i*, *n*\) – Returns a substring of the given string *s*. Parameter *i* gives the zero-based index of the beginning of the resulting substring and parameter *n* gives the length of the resulting substring. If *i* is not between 0 and **length(***s***)** or $n < 0$, the function returns the empty string. If $n >$ **length(***s***)** $- i$, then the function returns all the remaining characters of *s* from *i*-th character.

4.9 Embedded Class `Object`

This class defines the following methods:

- **string toString\(void \)** – Gives the chunk id that represents the object converted to **string**.
- **string getClass\(void \)** – Returns the string with the name of the run-time class of this object.

4.10 Implementation Notes

The design and the implementation of the compiler are completely up to you. Since extensive amount of work is necessary in order to create a compiler and considering the trends in modern compiler design, we recommend using as much automatic and generation tools that support the creation of the compiler as possible. More specifically, consider using 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 and Python library of version 4.7.2 can be accessed directly on Merlin server in directory `/pub/courses/vyp/`). In Python, consider using PLY. If you would like to use other tools, contact the project consultant for the approval.

5 Examples

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

5.1 Iterative Factorial Computation

```
/* Program 1: Iterative Factorial Computation */
void main(void) { // Program Main function
  int a, res;
  print("Enter an integer to compute its factorial:");
  a = readInt();
  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 */
void main(void) {
  int a; int res;
  print("Enter an integer to compute its factorial:");
  a = readInt();
  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 n * temp_result;
} // end of factorial
```

5.3 Program using Strings and Embedded Functions

```
/* Program 3: Program using Strings and Embedded Functions */
void main(void) {
  string str1, str2;
  int p;

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

  str1 = readString();
  while ((subStr(str1, p, 1)) != "") {
    p = p + 1;
  } // end of while
  print("\nThe length of \"", str1, "\", is ", p, " characters.\n");
} // end of main
```

5.4 Program with Classes and Objects

```
/* Program 4: Program with Classes and Objects */
class Shape : Object {
  int id;
  void Shape(void) { print("constructor␣of␣Shape"); }
  string toString(void) { return "instance␣of␣Shape␣" + (string)this.id; }
}
class Rectangle : Shape {
  int height, width;
  string toString(void) { return super.toString()
                              + "␣-␣rectangle␣" + (string)(this.area()); }
  int area(void) { return this.height * this.width; }
}
void main(void) {
  Rectangle r; r = new Rectangle;
  r.id = 42; r.width = readInt(); r.height = readInt();
  Shape s; s = r;
  print(s.toString());
} // end of main
```

6 VYPcode: Target Language Specification

The following two sections describe the target language (VYPcode) for the compilation of VYPlanguage. In the last section, the interpreter of VYPcode that is used for the evaluation of the generated code is documented.

6.1 General Information

VYPcode is the target language in VYPa Compiler project. It is close to typical assembly languages with the additional support for complex data structures such as structures, array, and objects. VYPcode and its interpreter (vypint) support

- 8 universal 64-bits registers (`$0` to `$7`); the maximum number of registers can given to the interpreter by a command-line option; the compiler can assume by default 8 registers.
- program counter (register PC)
- heap memory organized by chunks; chunk is a group of items (every item corresponds to a 64-bit word); chunks are addressed by a unique identifier
- the memory for the program is separated and inaccessible from VYPcode
- data stack is present (grows upwards; special register $SP (stack pointer)) and it is separated from the heap memory

6.1.1 Memory Model

- word length = 64 bits (the size covers IEEE single-precision float as well);
- number of 64-bits registers configured by a command-line argument of the interpreter --regs=*N*, where *N* is min. 1, default 8, syntax: $0, $1, $2, . . . , $N-1;

- special, user-mutable $SP register (stack pointer; refers to the top of the stack);
- special, user-immutable PC register (program counter), in fact, PC points to the next instruction to execute, so it is updated as soon as possible (after the decoding of the current instruction). PC is a reserved word in VYPcode so it cannot be used as an alias;
- stack = continuous memory of words for local variables, parameters, function frames (saving registers etc.); the words of the stack are indexed from the range $\langle 0, M-1 \rangle$ where the stack size M can be given by a command-line argument of the interpreter --stack=M ($M \geq 1$ and default value is 65535 words);
- heap of chunks = associative memory indexed by chunk identifiers (abbreviated as id); each chunk consists of (1) internal id, (2) number of items (size), and (3) consecutively saved items (potentially resizable); each item is a word, chunk id is an integer value that is always greater than 0, so 0 can be used as undefined chunk id.

Addressing:

- Register Addressing (reg) – the value of the operand that is in the register (e.g. `ADD` $0, $1, $2)
- Immediate Addressing (imm) – the integer or floating point literal or constant expression that describes the value so it is used only for source operands
- Indirect Stack Addressing (stack) – uses the value of the operand as an offset in the stack (e.g. `CREATE` [$1-1], 4); syntactically, it consists of a pair of square brackets enclosing a simple expression. A *simple expression* is, (1) a constant or (2) a register or (3) a register +/- constant)

6.1.2 Literals

- Integer (type: 64-bit int): supports decimal and hexadecimal format (0x0 - 0xFF...FF)
- Float (64-bit): supports decimal (decimal point is mandatory), scientific (e.g. 1E10), and hexadecimal (precise, %a) format
- String literal consists of *printable UTF-8 characters* (including escape sequences) that are enclosed by double quotes (`"`, UTF-8/ASCII value 34) where *printable UTF-8 character* is a character with UTF-8/ASCII value greater than 31 (apart from 34) or an escape sequence. VYPcode string literal is compatible with VYPlanguage string literal (see Section 3.1). Available escape sequences are as follows: **`\n`**, **`\t`**, **`\\`**, **`\"`** and of the form **`\x`*`hhhhhh`*** where *hhhhhh* is a hexadecimal number of exactly 6 hexadecimal digits. The meanings of these escape sequences correspond to the same escape sequences in C language. The internal representation of a string literal is a chunk of the given size (the size held by the chunk), each codepoint (typically represents a character) is stored in one word/item in UTF-8 encoding.
- *Name*: An alphanumerical string starting with a letter. Letters are from {`A` – `Z`, `a` – `z`, `.`, `:`, `@`, `_`}. Names are used in labels and named constants.

  Constant is a literal or named constant (see below).

6.1.3 Sections

Use the # (hash) or ; (semicolon) to comment out the rest of the line.
The *header* consists of the following three optional commented lines:

1. the "sh-bang" line (e.g. `#! /bin/vypint`);
2. the version of VYPcode (i.e. `# VYPcode: 1.0`); and
3. the comment with authorship (i.e. `# Generated by: xnovak00`

After the header, there is a *constant section* followed by an *instruction section*. The constant section consists of the definitions of *alias*es and *named constant*s. The alias is a user-defined name for a register and it has to start with $. For instance, `ALIAS FP $0`. We refer to the defined alias with the preceding $ symbol (e.g. `MOVE $FP, 10`). The named constant is a user-defined name for a constant expression, e.g. `CONSTANT FRAME_SIZE 4*INT.SIZE`. The name must be unique with respect to the other named constants and labels.

A *constant expression* is an arithmetical or string expression with a constant result. It allows basic binary operators (+, -, *) for numeric literals with usual meanings and precedences, parentheses, literals and other already defined named constants. The only provided operator for strings is concatenation (symbol +).

6.2 Instructions

Every instruction consists of an operation code (opcode) followed by comma-separated operands (comma can be replaced by at least one space or tabulator).

The definition of a label starts with keyword `LABEL` and continues with a name. It has to be unique with respect to the other labels and named constants.

The available addressing mode is specified in the parenthesis behind the operand name in the following specifications. Most of the instructions have a destination operand only with reg addressing and source operand(s) with reg/stack/imm addressing.

Chunk Manipulation:

- `CREATE` *dst*, *size* – destination register *dst* (reg) for id of a new chunk of the given *size* (stack/reg/imm) (*size* ≥ 0);
- `COPY` *dst*, *id* — do a shallow copy of the given chunk *id* (stack/reg/imm) and save the id of the new chunk in *dst* (reg). If the chunk *id* is invalid, exit interpretation with an error;
- `GETSIZE` *dst*, *id* — destination register *dst* (reg) for the size of the given chunk *id* (stack/reg/imm); -1 if *id* is invalid;
- `GETWORD` *dst*, *id*, *index* — assign the word value of an item given by the zero-based *index* (stack/reg/imm) of chunk *id* (stack/reg/imm) into the destination register *dst* (reg);

- `RESIZE` *id*, *newsize* — change the size of the existing chunk (given by *id* (stack/reg/imm)), the size (stack/reg/imm) can be greater or even smaller than the current size of the chunk. The instruction preserves the content of the re-sized chunk up to *newsize* (if less than the original size). If the chunk *id* is invalid, exit interpretation with an error;
- `SETWORD` *id*, *index*, *value* — set *value* at zero-based *index* in the chunk *id* (all addressed by reg/stack/imm);
- `DESTROY` *id* — destroy and deallocate the chunk of the given *id* (reg/stack/imm); if the chunk *id* is invalid, do nothing.

Control Flow Instructions:

- `CALL` *pc*, *where* — stores incremented PC[6] into *pc* (stack) and if *where* is a label, then jumps into the label; otherwise *where* (reg/stack/imm) is a chunk id of a string that contains the label to which it jumps (dynamic jumping); if the string is not a valid label, it is a run-time error;
- `RETURN` *pc* — restore PC from *pc* (reg/stack/imm);
- `SET` *dst*, *value* — copy *value* (reg/stack/imm) into *dst* (reg/stack);
- `JUMP` *label* — jump on the given *label* (by its literal, no jump address);
- `JUMPZ`/`JUMPNZ` *label*, *src* — if *src* (reg/stack/imm) is zero/non-zero value, then jump to *label*.

Input and Output: Input and output instructions operate in UTF-8 encoding.

- `READS` *dst* — read a new string (the whole line from `stdin`, the end of line (EOL) is not included) into a new chunk returned in *dst* (reg);
- `WRITES` *id* — write the string stored in chunk *id* (reg/stack/imm) into `stdout` (no additional EOL);
- `READI`/`READF` *dst* — read an integer/float as the first value from `stdin` (read away the rest until the end of line) into the given register *dst* (reg);
- `WRITEI`/`WRITEF` *src* — write an integer/float given by *src* (reg/stack/imm) into `stdout` such that float is formatted using %a (precise hexadecimal format).

Arithmetic, Relation, Logic, and Conversions: If the instruction works with floating point numbers as input operands, these operands are interpreted as floats. Since the addressing of destination and source operands is usual (dst: reg, src1 and src2: stack/reg/imm), they are omitted:

- `ADDI`, `SUBI`, `MULI`, `DIVI` — arithmetic operators with integers;
- `ADDF`, `SUBF`, `MULF`, `DIVF` — arithmetic operators with floating point numbers;
- `LTI`, `GTI`, `EQI` — less than, greater than, and equivalence for integers (results: 0 = false, 1 = true);

[6] Warning: VYPcode can store and load the program counter but the modification of such value has undefined behaviour and can result into weird behaviour of the program.

- `LTF`, `GTF`, `EQF` — less than, greater than, and equivalence for floats (results: 0 = false, 1 = true);
- `LTS`, `GTS`, `EQS` — less than, greater than, and equivalence for chunks with strings (results: 0 = false, 1 = true);
- `AND`, `OR`, `NOT` — Boolean operations (non-zero operands are understood as true; zero as false; results: 0 = false, 1 = true), NOT takes only one source operand;
- `INT2FLOAT`/`FLOAT2INT` *dst*, *src* — converts integer to float or trims float to integer.
- `INT2STRING`/`FLOAT2STRING` *dst*, *src* — converts integer/float number to string into a new chunk returned in *dst* (e.g. -15 is converted to string "-15" and 3.14 is converted to string "0x1.91eb851eb851fp1").

Debugging instructions: These instructions work only when the interpreter is not silent (see `--silent` command-line option).

- `DPRINTI`/`DPRINTF` *src* — prints *src* (stack/reg/imm) as an integer/float (both decimal and precise hexadecimal representations) into `stderr`;
- `DPRINTS` *id* — prints a string stored in chunk *id* (stack/reg/imm) as a UTF-8 string into `stderr`;
- `DUMPREGS` — prints the current contents of all registers into `stderr`;
- `DUMPSTACK` — prints the current contents of the used items of the stack into `stderr`;
- `DUMPHEAP` — prints the current contents of all chunks on the heap into `stderr`;
- `DUMPCHUNK` *id* — prints the current content of the chunk given by *id* (reg/stack/imm) into `stderr`.

6.3 Interpreter

To test and process a source code in VYPcode, there is a command-line interpreter (`vypint`):

```
java -jar vypint-1.0.jar src.vypcode < input > output
```

The behaviour of the interpreter can be modified by the following command-line arguments:

- `--help` Print out this help information how to use the interpreter.
- `--regs=`*N* Configure the number of available general purpose registers ($N \geq 1$).
- `--stack=`*N* Configure the size [words] of available stack ($N \geq 1$).
- `--verbose` Activate the debugging mode of the interpreter so it prints details about every executed instruction.
- `--silent` Activate the production mode, so the debugging instructions will not print anything.

If the interpretation runs without an error, it returns `0` (zero). The return codes for error states follow:

- 20 – invalid command-line arguments (e.g. bad source filename);
- 21 – lexical or syntax error of VYPcode source code;
- 22 – semantic error of VYPcode source code (e.g. missing label);
- 25 – runtime error – invalid or non-existing label in the given chunk;
- 27 – runtime error – division by zero;
- 28 – runtime error – invalid memory access (e.g. non-existing chunk or bad offset on the stack);
- 30 – internal interpreter error – the error is not influenced by the source program (e.g. not enough memory).

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 submitted only by the team leader. All submitted 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 before the given date (see above). Failure to do so will lead to project being considered as not submitted at all or at least final score will be penalized proportionally to the length of the delay. Similarly, any form of plagiarism will lead to project being scored as 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, PLY, ...) 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 can be implemented using **any programming language** (e.g. C, C++, Java, Python[7]) that can be compiled and run on the `Merlin` server (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 are 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 `vypcomp`, which has one mandatory and one optional argument, as is described in Chapter 2.

8.2 Text Documentation

A 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 must** contain:

- 1st page: full names 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, methods and tools.
- Description of your back-end implementation - design, implementation, intermediate code, symbol table, special techniques used, optimizations, and 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 another environment explicitly allowed by the project consultant.
- Literature used, including citations of any non-original materials (pictures, statistics etc.).

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

The documentation shall not:

- Contain a copy of any part of the project specification or any text, pictures[8] 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 sufficiency, but not an excessive number of comments will be also considered as a part of the documentation evaluation.

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

Every error message that arises from the run of the program must be outputted to the standard error output. The program shall not output any characters or text to the standard output during its execution, except the text described by the control program written in VYPlanguage. Basic tests will be automatically performed using scripts that will gradually run a set of test programs (compilation of the program with your compiler and interpretation of its resulting target code by vypint) and compare the output of your target code interpretation 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 VYPa 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.).

A 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. **A 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 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 VYPa lectures, VYPa 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 extensi-

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

ons that you have implemented. Each line shall contain one identifier followed by a UNIX style newline character (i.e. ⟨LF⟩ sign).

During the semester the list of registered extensions and their identifiers can be gradually expanded. See VYPa 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 Extensions that Can Be Rewarded by Extra Points

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

- FLOAT: Support for primitive data type **`float`** for C-double precision floating point numbers based on C-like syntax including the reading of floats. The generated target code will contain floats only in hexadecimal precise format (+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).
- 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[9] (+0.5 point).
- INITVAR: Initialization of a (instance) variable with an expression during its definition (+0.5 points). In case of an instance variable, the initialization happens before the constructor call of the corresponding class but after the constructor call of its superclass.
- 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 (+0.5 point).
- OVERLOAD: Overloading of user-defined functions and methods (including constructors such that creation of instances with parametrized constructor has the syntax from Java where the empty brackets can be omitted) (+2.0 points).
- SHORTEVAL: Support the evaluation of conditional expressions using the *short evaluation* (+1.5 points).
- VISIBILITY: Support visibility modifiers of class members. There can be at most one modifier in front of each member definition. Keyword **`public`** is the implicit choice (see Section 4.4). Keyword **`protected`** allows access only to methods from the same class and its subclasses. Keyword **`private`** restricts the access only to the methods of the same class. A violation of a visibility rule leads to semantic error 14 (+1.0 point).
- ...

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