VYPa 2018 - Project Documentation

December 23, 2018

Authors and points division:

- Roman Andriushchenko (xandri03) - 100
- Dmytro Zanora (xzanor00) - 0

Implemeted extensions: -

Implementation

The compiler is implemented in the Java language. To simplify the creation of the parser, automatic generation tool ANTLR[1] was used. The tool provides classes that allow the (simulation of) construction of the derivation tree. The grammar for this tool describing the grammar of the VYPlanguage can be found in `Grammar.g4` file. At the start of the compilation, this file is processed and the parser code is created within `parser` package. This package is used in the `main` package that contains the (manually written) source code of the VYPlanguage compiler. The following text describes the process of compilation of a program in VYPlanguage.

If the input was successfully read, lexical and then syntactic analysis are performed. This is achieved by traversing the derivation tree while replacing error listeners with listeners that return the corresponding exit code. Then, we collect names of classes, names of their bases and check whether the class hierarchy is valid, i.e. does not contain loops. Now we know all data types that can be used within a program and can collect headers of global functions.

Next, we collect all instance attributes and headers of methods. In the latter case, implicit "this" parameter of the corresponding type is inserted as a zeroth parameter. If the explicit constructor for a class is missing, we create one with empty body. Finally, for each constructor we insert explicit commands for the initialization of instance variables.

At this point it is appropriate to mention the use of `SymbolTable.java` which is used as a (separate) table of classes, table of attributes/methods for classes or a table of variables for functions. In the latter case, the table represents the scope of the body of the method and also contains function parameters. In case this body contains additional blocks of commands (conditional/iteration statements), a new scope is created having the previous scope as its 'parent': the new scope will contain overlapped variables and non-overlapped ones can be found in the parent scope.

Now that we know definitions of all classes used in the program and know the signature of each function/method, we are ready to process the body of each function/method. There is nothing worth mentioning about this process: table of classes will assert existence of the corresponding class; table of global functions will allow to check whether a function call is correct (while taking into account

[1] `https://www.antlr.org/`

class hierarchy and subsumption rule); table of local variables for a given method will allow to assert non-existence of a newly declared symbol or a type compatibility when existing one is used.

Now that the parsing is successfully finished, we can generate the code. Before this, to each attribute, method or a local variable we assign an integer index which will be used to access the contents of the corresponding entity:

- index of an attribute is a positive $(1, 2, \dots)$ index to the object structure; index 0 will contain a pointer to the VMT of the object,
- index of a method $(0, 1, \dots)$ is an index to the VMT; if a method overrides existing method in the superclass, this method will have the same index as the overridden method,
- index of a formal parameter $(-1, -2, \dots)$ and index of a local variable $(1, 2, \dots)$ is an index to the stack.

Note that during code generation we also generate comments to improve its readability for the process of debugging. This behaviour is parametrized and can be easily suppressed. The following text describes the contents of the generated program.

We begin by creating a table of VMTs for each of the classes and storing the pointer to this table to a designated register. Individual VMT is nothing but a table of function labels of the form `class::method` that will be generated later in the code. We proceed by generating a code that calls `main` function and then jumps to the end of the program, finishing the computation. We then proceed by generating the code for each of the global functions and the code for each of the class method. Note that we do not generate the code for `print`, `readInt` or `readString` global functions: their calls are replaced by the sequence of `WRITEI`/`WRITES` commands, by the `READI` or by the `READS` command, respectively.

Compilation

Although primarily the project utilises `ant` tool to compile the project using `build.xml` file, additional `vypcomp` script and `Makefile` were created to satisfy the required compilation/execution procedure. The program can be normally compiled on `merlin` server and only the existence of `/pub/courses/vyp/antlr-4.7.1-complete.jar` file (for ANTLR generation) is required.

Work division

Due to personal reasons, Mr. Dmytro Zanora decided not to participate in this project and therefore Mr. Roman Andriushchenko is the sole author of the source code as well as this documentation. This fact is reflected in the points division.