BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

VYPa: Compiler Project Documentation

FLOAT
IFONLY
INITVAR
VISIBILITY

Peter Kubov, xkubov06 (50%)
Richard Mička, xmicka11 (50%)

22. 12. 2020

Front-end implementation

The front-end of the Vypcomp compiler was created in C++17 with usage of Bison and Flex tools. We took advantage of the object-oriented extensions of these tools and created stand-alone modules doing specific translation features and communicating with each other in a standard object-oriented way. In the next sections will be introduced each module of the compiler's front-end. The main module that interconnects all compiler parts and triggers compilation is located in `src/vypcomp/vypcomp.cpp`

Lexical Analysis

Lexical analyser was designed and generated using Flex. Flex is capable of automatic grammar generation and supports modern object-oriented module design in C++[1]. For each language feature a regular expression was provided and upon match is converted into a specific language token. Output of lexical analysis is therefore a chain of tokens. These token objects are defined in parser and have a form of a C++17 std::variant[2].

Implementation files:

- `include/vypcomp/parser/scanner.h`
- `src/parser/scanner.l`

Syntax Analysis

Module of syntax analyser was designed and generated using GNU Bison. For this project we chose LALR(1) context-free grammar that was defined using Bison's syntactic rules. As a result of the parser generation a special class of parser was created that was used to analyse input as a result of communication exchanged by objects. The syntax analysis is triggered in the main module that interconnects all compiler modules.

Implementation files:

- `include/vypcomp/parser/parser.h`
- `src/parser/parser.yy`

IFONLY Extension

As the VYPa20 language requires all if statements to have curly brackets to start block we did not have to solve pairing and every else statement is unambiguously assigned to the closest if statement.

[1] https://ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html_node/flex_19.html
[2] https://en.cppreference.com/w/cpp/utility/variant

Semantic Analysis

For the purpose of semantic analysis a ParserDriver interface was designed. This interface defines methods that are triggered during syntax analysis and when implemented can activate semantic analysis response. As the compiler must support usage before declaration, two classes implementing the ParserDriver interface were provided. Objects implementing these classes are then connected sequentially to process input. The first class is called IndexParserDriver and is responsible for indexing namespaces and initializing symbol tables with correct references. Objects of this class ignore semantic analysis of expressions and statements as they may contain references to currently undefined names. The driver of this kind is mainly responsible for redefinition checks. At the end of processing the driver returns an initialized global symbol table that is inserted into the object of class ParserDriver. This is the main semantic analysator as all named references must be available at this point. Every semantic action is implemented in a dedicated method that is invoked in a syntax parser generated by Bison.

Implementation files:

- `include/vypcomp/parser/parser.h`
- `include/vypcomp/parser/indexdriver.h`
- `src/parser/parser.yy`
- `src/parser/parser.cpp`
- `src/parser/indexdriver.cpp`

Intermediate Representation

To match our Object-Oriented model we designed each instruction as specialization of abstract class Instruction. Instructions are chained into a list that can be viewed as a list of atomic steps of the VYPa20 program. The same process applies to the expressions. The only difference with expression is, that expressions are organised into tree structure instead of list. This design was chosen as it is much easier to apply syntax and semantic analysis this way. Generation of expressions is afterwards done by post-order traversal of the expression tree.

Implementation files:

- `include/vypcomp/ir/expression.h`
- `include/vypcomp/ir/instructions.h`
- `include/vypcomp/ir/ir.h`
- `src/ir/expression.cpp`
- `src/ir/instructions.cpp`
- `src/ir/ir.cpp`

Symbol Table

The symbol table is implemented as a STL Map that has internal implementation of a balanced AVL tree. Symbol table consists of three kinds of items - class, function and variable, each with appropriate attributes. During parsing symbol tables are organized into a stack. New table is created and inserted into the stack when the parser runs into a new block. After the block is parsed, the table is removed from the stack. This enables the parser to stack namespaces and take care of redefinitions.

Implementation files:

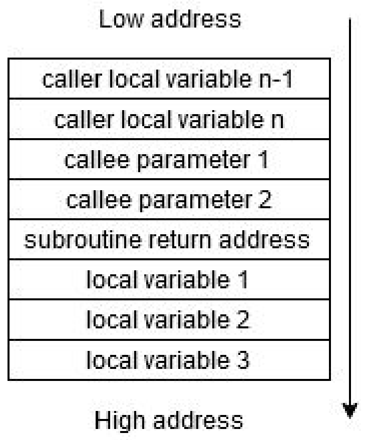
- `include/vypcomp/parser/symbol_table.h`
- `src/parser/symbol_table.cpp`

Back-end implementation

Our back-end is implemented inside the Generator class. It operates on a single input from the compiler driver. The compiler driver retrieves the final global symbol table from the front-end's semantic analyzer ParserDriver, constructs the Generator and then relays the symbol table to the Generator's generate routine. This routine iterates over all global symbols, which can either be functions or classes in their intermediate representation. Generator can produce annotated target code with helpful labels if the verbose option is provided to the compiler. Every program starts with a stub that creates tables for virtual functions (more details in the Class code generation section) and then calls the main entry point. After the program returns from the main entry point, it jumps to the end of the program which is always labeled "ENDOFPROGRAM". This way it does not impose restrictions on the order of function generation.

Function code generation

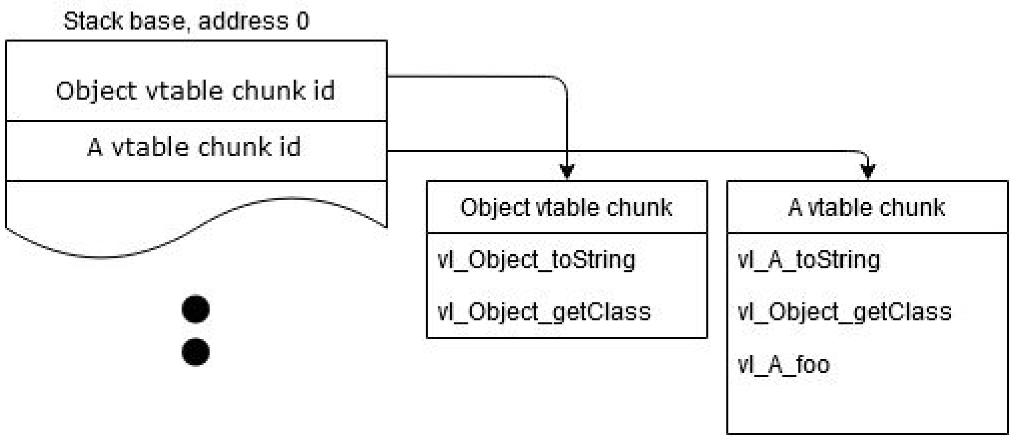
All functions are generated as independent units. Every function accepts its arguments on top of the stack, which are immediately followed by the return address. Function can optionally use local variables, which can reside on the top of the stack. In that case, it moves the `$SP` register further creating more space for the local variables. The stack layout for a function with 2 parameters and 3 local variables is shown in the picture below. Callee is responsible for stack cleanup in our chosen calling convention. Subroutine epilog thus consists of shifting the stack pointer by the number of local variables, getting the return address, moving the stack pointer by the number of parameter + 1 (for the return address). The return register of a subroutine was chosen to be `$0`. At the start of function generation, the generator does one simple pass through the instructions to gather information about required



temporary variables. If some instructions request additional temporary variables, they are added to the list of local variables and are generated and used exactly the same way an ordinary local variable would. If there is a scoped variable shadowing a variable with identical name in the outside scope, they are distinguished as unique objects with distinct addresses in the Function's IR.
On the second pass, the generator iterates over all instructions of a function generating target code on the fly. Special attention is given to branch and loop instructions. These contain references to additional standalone basic blocks which are generated into string streams, which are then interconnected with the surrounding basic block through either conditional or unconditional jumps as required to achieve the desired loop or branching behavior. Expressions are represented as generic trees. The so-called 'simple' expressions are SymbolExpression and LiteralExpression. Each of which require just a single register load instruction "`SET`". All other expressions can contain other sub-expressions and therefore require a temporary variable location to store their value. Therefore during the first generator pass, all non-simple expressions request a temporary variable which is then added to the local variables of the function.
Built-in functions such as "`length`", "`readInt`", "`readFloat`", "`readString`" etc. are generated statically for every program and they are already contained in the generator in the target code form inside "`generate_builtin_functions`" function. One additional function which is not among the required built-in functions is generated, "`addStr`" subroutine. This function is used to implement string concatenation which is a part of the core language. All other functions use labels with a prefix "`vl_`" to avoid possible conflict with automatically generated labels of the "`addStr`" routine, branching and looping instructions or object type cast. Specially treated is the function "print", which has an unlimited number of primitive type parameters. The semantic check of this function's parameter count is therefore disabled. The call to this function is treated differently as well. Instead of the usual preparation of parameters, pushing to the stack and calling a subroutine, print function evaluates the parameters from left to right and calls the corresponding "`WRITEI/WRITEF/WRITES`" instructions for each one of them based on their type.

Class code generation

As mentioned above in this chapter's introduction, virtual function tables (shortened as vtable) which are responsible for creating overriding behavior in methods are generated at the start of the program at the stack base before entering main. Each class is assigned a unique id which corresponds to its vtable's stack address. Vtable's stack address contains a chunk id which then contains chunk ids of label names of functions. This stack address is then assigned to each object of the class's type in its constructor. To properly generate a vtable, methods of the whole class hierarchy from the currently generated class up to the Object class have to be enumerated with ids. These represent the methods position in the vtable's chunk. An example of 2 vtables of class A, which is derived from Object, and class Object are shown in the picture. Class A overrides the inherited method "`toString`" from Object, thus its label is "`vl_A_toString`", but does not override the method "`getClass`", therefore the label of this method remains the same. Once a method call expression is encountered, the target method is determined by reading each object's vtable pointer and getting the correct method label.



Constructor is separated into two parts, the constructor responsible for creating the object chunk and then the constructor body, which can be defined by the user. Every object's first attribute is the stack address of its class's vtable pointer. The second attribute is the run-time type attribute assigned to it. This attribute is used to validate real object type in object casts and in "`getClass`" built-in method. After constructing the object and setting these two special attributes, the parent constructor chain is invoked. After the parent constructors are finished, the object's attributes are set to their assigned values from INITVAR extension or are initialized to 0. Methods itself are generated identically to ordinary functions with the only difference being the label name generation. Instead of the "`vl_`" prefix followed by the function name, methods use the naming: "`vl_`" prefix, class name, which is followed by the method name. The method call context object is treated as an implicit first parameter of all methods and is pushed automatically into the proper stack position before the method call is executed.
oop oriented expressions:

Testing

During development we designed and implemented two sets of testing frameworks for specific parts of our compiler. For the front-end testing we used Google GTest[3] and created an extensive set of tests that were designed to ensure correct behavior of syntactic and semantic analysis.

[3] https://github.com/google/googletest

To test correct behavior of code generator we created another set of tests - regression tests using python framework *unittest*[4]. These tests not only test the generator but also correct communication of all implemented parts of the compiler. Having these testing framework allowed us to incrementally create test cases of supported features:

- an input file containing the VYPa20 source code
- input provided to the VYPcode interpreter as stdin
- expected output of the compiled source code
- optional expected non-zero return code of the compiler
- optional expected non-zero return code of the interpreter

We did behavioral testing by creating test cases in VYPa20 source code and then compiling them into VYPcode. On the generated output we used VYPInt interpreter and tested whether output has the same semantics as was originally written in the test case. The source files of test cases are located in `"tests/compiler_cases"` directory. Test cases are written in python and can be found in the following file: `"tests/compiler_cases/test_cases.py"`. To execute these tests you have to be in `"tests"` directory and run:

```
python3 compiler_tests.py <vypcomp_path> <vypint_path>
```

Work Distribution

During the semester we organized our work by communication on a regular basis. As we designed the compiler into modules we divided work on each module evenly in terms of workload, time difficulty and lines of code in the following way:
Peter's responsibilities:

- Scanner and Parser generation.
- Instructions intermediate representation.
- Module parsing, statements.
- Semantic Analysis implementation.
- Symbol table implementation.
- Unit testing.

Richard's responsibilities:

- Expression intermediate representation.
- Expression parsing.
- Semantic analysis of expressions.
- Code generation, OO model implementation.
- Behavioral testing, regression tests.

We synchronised our work using the git version control system and applied standard development techniques to ensure productive workflow. Based on the even distribution of work we decided to split points in the only fair way:

- Peter Kubov 50%
- Richard Mička 50%

[4] https://docs.python.org/3/library/unittest.html