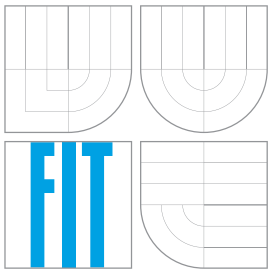
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PŘEKLADAČ A INTERPRET JAZYKA VYP08

COMPILER AND INTERPRETER OF VYP08 LANGUAGE

AUTOR PRÁCE
AUTHOR

KAMIL DUDKA

BRNO 2008

License

Contents

Chapter 1

Overview

`vyp08` is compiler and interpreter of `VYP08` language[1] developed as a school project for VYP course. Its input is a program in `VYP08` language. There is no equivalent output program written to file and the compiled program is given directly to the interpreter instead (in an internal representation based on `vyp08` object model).

The compiler part of this program is heavily based on *FLEX/Bison* equipment. FLEX is used to generate the core part of `scanner` module and Bison is used to generate the core part of `parser` module. FLEX/Bison utilities have been used for decades in many successful projects and they can significantly speedup the development of projects like this.

The main goal of this project was development of generic and reusable compiler modules which can be used later again in similar projects. The key part of design is therefore object model based on modern programming techniques like design patterns [2]. As the components are intended to be reusable, all public classes are well documented in API documentation (generated by Doxygen[2]).

In the following chapter are briefly described utilities FLEX and Bison. In the chapter 3 is the object model explained step by step. In the chapter 4 are a few hints about implementation, build and testing. And finally in the chapter 5 is summarized the contribution of this work.

[1] `VYP08` language definition can be found in the task description, nowadays available at https://www.fit.vutbr.cz/study/courses/VYP/private/projekt/vyp2008.pdf

[2] source code documentation generator tool, available at http://www.stack.nl/~dimitri/doxygen/

Chapter 2

Theoretical part

In this chapter are briefly described utilities FLEX and Bison. The goal is to explain why are these utilities useful for development of compiler like `vyp08`. Huge amount of documentation, tutorials and FAQs can be found in referred (mostly on web published) materials.

2.1 FLEX

FLEX (*Fast LEXical analyzer generator*) is a tool for generating scanners [3]. On its input is an program in the FLEX language, roughly speaking a set of regular expressions defining desired lexical units. Output of FLEX is a program in C/C++ which can read these lexical units.

The generated scanner is based on FSM (*Finite State Machine*). The family of languages accepted by regular expressions is equivalent to family of languages accepted by FSM [9]. Each regular expression can be converted to equivalent non-deterministic FSM, each non-deterministic FSM can be converted to deterministic FSM and each deterministic FSM can be converted to minimal FSM. In this way is the scanner generated by FLEX from a set of regular expressions on its input.

FLEX manual which includes a lot of examples and FAQs, can be found at http://flex.sourceforge.net/manual/.

2.2 Bison

As FLEX is generator of scanner, Bison[1] is generator of parser. On its input is an program in the Bison language, roughly speaking an CFG (*Context-Free Grammar*) which defines the language accepted by parser being generated. Output of Bison is an program in C/C++ which can read the language defined by the CFG.

[1] Bison is based on YACC (*Yeat Another Compiler-Compiler*, its predecessor) and is backward compatible with it.

In the default configuration the *LALR(1)* parser is generated. LALR1 is a kind of very strong LR-parser, which can read any language acceptable by deterministic PDA (*PushDown Automata*) [4]. It is possible to generate GLR (*Generalized LR*) parser by Bison. This kind of parser can read languages defined by general CFG, but it is more complex to use and even more complex to run.

Bison manual which includes a lot of examples and FAQs, can be found at http://www.gnu.org/software/bison/manual/html_mono/bison.html.

Chapter 3

Solution design

In this chapter is presented the object model of `vyp08`. The program consists of a few separated design modules (which partly corresponds to C++ modules being built). The modules are intended to be independent on each other as much as possible. Each module has simple and generic interface which hides its implementation. This good manner makes it possible to take one module and replace it by another one with completely different implementation.

3.1 Scanner interface

As it has been already told, the core part of scanner is generated by FLEX utility. But this is not reason to make scanner interface dependent on FLEX. Therefore the FLEX is used only to generate part of implementation of this module. The public interface is much more generic. We can take (already compiled) `vyp08` project and replace scanner module by another implementation of scanner, which does not know anything about FLEX. Moreover we do not have to recompile the rest of `vyp08` project – link target binary is the only thing we have to do[1].

The interface of `scanner` module is pretty simple – there are two value types, two interfaces (pure virtual classes in C++ terminology [6]) and one static-only factory [2]. Enumeration type `EToken` represents scanner token type (e.g. `T_ID` or `T_ELSE`). Value type `Token` couples token type with its value (which makes sense only for tokens holding a value).

The root level interface `IErrorSensitive` defines simple method `hasError`. This interface is used as base class for all fundamental compiler parts which are sensitive to errors. The interface `IScanner` (based on `IErrorSensitive`) defines one more method to read the tokens from input program. Static-only class `ScannerFactory` is responsible for scanner object creation. The whole scanner interface is well documented in the API documentation mentioned above.

[1] It is also recommended to run the test suite which is distributed with compiler, to check if compiler works.

3.2 Scanner implementation

The implementation of scanner consists of two modules, one is generated by FLEX and one is written manually as a sort of wrapper around the code generated by FLEX. Scanner generated by FLEX is used to read all tokens except keywords (and keyword-like operators). The keywords are read as identifiers by FLEX scanner and handled later on.

FLEX can generate a legacy code for old program written in C language, but newly [8] it can generate a C++ compatible code[2] as well. Then the Flex scanner is generated as an ordinary C++ class and this approach was chosen for `vyp08`.

The class generated by Flex is used as a base class for `PrivateFlexLexer` which implements *NVI* (Non-Virtual Interface [7]) over the FLEX scanner. Classes `FlexScanner` and `KwScanner` stand for a *decorator* [2] chain based on `IScanner` interface. The first one checks literals read by FLEX scanner and converts them to value. The second one converts identifiers representing a keyword, to tokens representing the keyword.

3.3 Parser

Analogous to previous module, parser's interface is not dependent on its implementation. The interface is based on callback as the compilation process is syntax-directed. Pure virtual class `IBuilder` defines the callback and it separates parser and builder modules. The parser does not know anything about builder and the builder does not know anything about parser. This pure virtual class is the only thing which is common for both modules.

Similar to scanner, the implementation of parser has two parts. The first one is generated by Bison and the second one is written manually as something like wrapper around the code generated by Bison. Both parts are connected by `IBisonListener` interface.

In the Bison input is only the `VYP08` grammar defined, there is almost no code inside. The action code consists only of invocation of appropriate `IBisonListener` method. Class `BisonListener` implements the `IBisonListener` interface and acts as bridge between parser generated by Bison and the builder.

3.4 Virtual machine

As the parser module stands for the core of compiler, the core of interpreter is a module called *virtual machine* (shortly *vm*). Each basic abstraction defined by `VYP08` language has its equivalent in the virtual machine. There is a whole hierarchy of run-time objects. On the low level there are simple value types and on the topmost level is the class `Vm` which represents the whole virtual machine. Class `VmRunner` acts as servant class [5] used to run

[2] The C++ support in FLEX has been not declared stable yet. Its future modification can break even the source code level compatibility.

virtual machine (to run an abstraction of compiled `VYP08` program). On the collaboration diagram 3.1 is visualized the situation from runner's perspective.

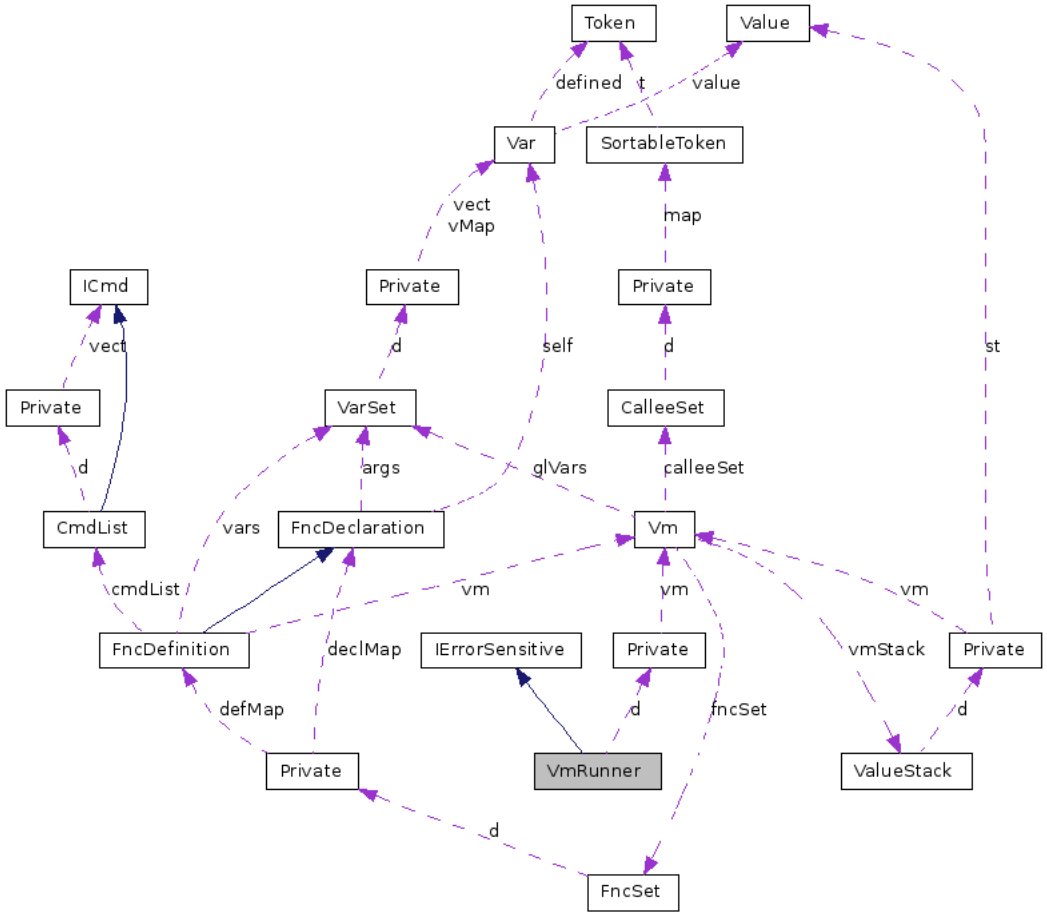


Figure 3.1: Collaboration graph of class `VmRunner`

Value type `Value` stands for a generic data type defined by virtual machine. Universal variable for global/local variables, function arguments and function return value is defined by value type `Var`. A smart container for such variables is the class `VarSet`. Declaration of function is represented by value type `FncDeclaration`. Function definition (value type `FncDefinition`) is directly inherited from type `FncDeclaration`. This inheritance makes comparison of two declarations or definition with declaration much more easy. Container `FncSet` acts as a common storage for `FncDeclaration` and `FncDefinition` objects.

All actions performed by the `VYP08` code are abstractly defined by interface `ICmd` (design pattern *command* [2]). A sequence of actions is therefore a container of objects which implements `ICmd` interface. This container (class `CmdList`) implements `ICmd` interface as well (design pattern *composite* [2]). Particular `ICmd` implementations are placed in separate module called *cmd*. The vm module does not know (and does not need to know) about these implementations.

The running `VYP08` program (strictly speaking its abstraction) needs some workspace – amount of memory to store temporary results etc. A stack was considered a suitable data structure for such purposes. Class `ValueStack` stands for a stack of `Value` objects used by particular `ICmd` implementations. The whole virtual machine is represented by value type (structure[3] in C++ terminology) `Vm`. Its fundamental members are:

- `glVars` - container for global variables
- `fncSet` - container for function declaration and definitions
- `vmStack` - object of type `ValueStack` (mentioned above)

3.5 Builder

Up to now it has been explained how does virtual machine works. There has been nothing told about construction of virtual machine equivalent to interpreted `VYP08` program yet. This is builder's responsibility.

Instance of builder is created by factory `BuilderFactory`. Builder implements the `IBuilder` interface (mentioned above) defined by parser. Each particular method implementing `IBuilder` interface builds a part of virtual machine corresponding to the event fired by parser. In the builder's public interface is one more factory class – `FncFactory` which creates the built-in function's declarations/definitions during virtual machine initialization.

Collaboration diagram 3.2 of class `Parser` shows the object model being used during the build of virtual machine. Parser uses instance of scanner object to read input and instance of builder object to build virtual machine equivalent to input program in `VYP08` language.

Factory class `CmdFactory` belongs to module cmd mentioned above and is used by builder to obtain `ICmd` implementations which perform requested action in the run time. The `ICmd` objects are put to appropriate `CmdList` object which represents a block of commands in the input `VYP08` program – either function body or part of `if`/`while` statement. The appropriate `CmdList` object is always the top of *block stack* which is maintained by builder. The static type checking is based on *type stack* which is maintained by builder as well.

[3]The only difference between class and structure is the default visibility from the C++ compiler's perspective. Therefore it has been called as class first to not frighten the reader.

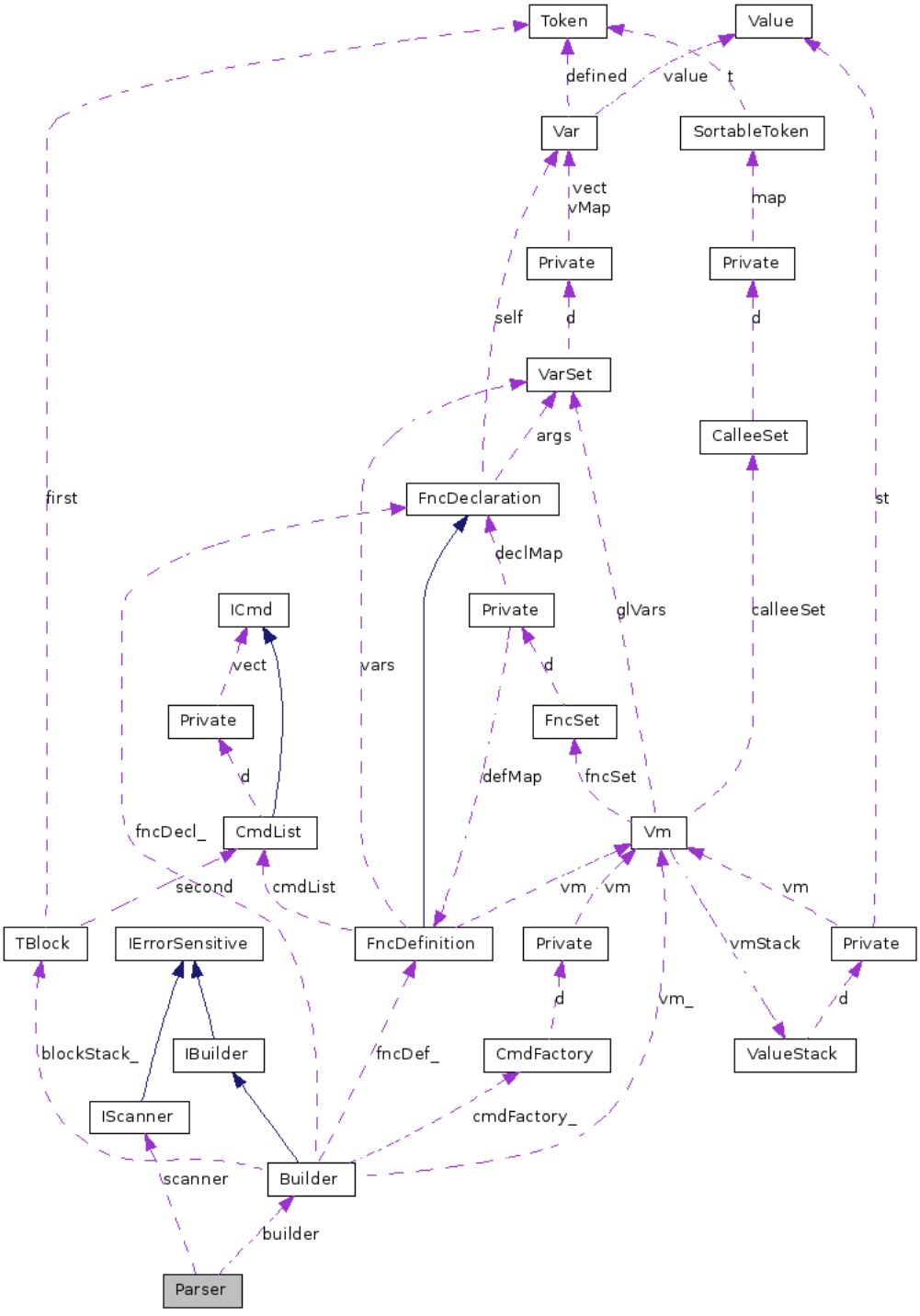


Figure 3.2: Collaboration graph of class `Parser`

3.6 Summary

Diagram 3.3 shows relations between modules. Note these dependencies are between module's implementations. Looking at module's public interfaces there are almost no dependencies between them.

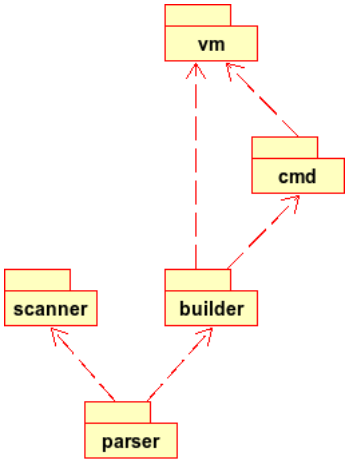


Figure 3.3: Modules dependency graph

Chapter 4

Implementation

`vyp08` is written in the C++ language. Not counting FLEX/Bison it depends on *Boost*[1] libraries. The library called `smart_ptr` provides various types of smart pointers [1]. Template `shared_ptr` from this library is used instead of native C++ pointer in certain places in the `vyp08` code. `shared_ptr` behaves as native C++ pointer, but in addition it counts references to pointed object and destroy the object automatically when the references count became zero.

Each module includes header `config.h` which consolidates various compile-time configuration options. Some diagnostic messages can be turned on/off within this file. In the `config.h` are also defined conditional macros as wrappers around some STL and Boost containers to work better with Doxygen.

4.1 Build from sources

Build of `vyp08` from sources is based on *CMake*[2] (the cross-platform, open-source build system). CMake checks availability of Flex/Bison utilities and required Boost libraries. It can also scan dependencies of compiled modules and automatically generate Makefile for the project.

By default `vyp08` uses colorized console output for `stderr`. You can switch it off by changing value of `CONSOLE_COLOR_OUTPUT` macro to zero in `config.h` (mentioned above) before build – this is an compile-time[3] option.

For successful build you need Flex, Bison, Boost and CMake. To build project from sources just type `make` in the project directory. You can also build this documentation (requires LaTeX) and API documentation (requires Doxygen). To do so, type `make doc`.

[1] available at http://www.boost.org/

[2] available at http://www.cmake.org/

[3] Implementing this as run-time option is a trivial change but it violates with task description as there can be no more command-line arguments.

4.2 Test suite

After successful build it is recommended to run the test suit distributed with compiler. It checks if `vyp08` works as expected. This is very useful in case of some future changes (e.g. bug fixes) to avoid regression. The test suit consists of 40 tests.

The first two are unit tests for scanner and parser modules. Both of them can be run in manual mode in case of debugging. The manual mode can be initiated by passing '`-`' as the command-line argument to the test binary. Then it reads the `VYP08` code from standard input and writes corresponding lexical (in case of scanner) or syntactical (in case of parser) units to standard output.

Remaining 38 tests work with the `vyp08` binary. Each test gives a piece of `VYP08` (valid or invalid) code to `vyp08` to execute. If the test requires some input, the test gives it to compiler's input. Then the test checks the exit code of `vyp08` and (optionally) its output with the expected one.

To run this test suite type `make check` in the project directory. If it succeeds, exit code is zero. If not, there are some additional information in the build directory.

Chapter 5

Conclusion

Implemented compiler and interpreter `vyp08` works as it was requested. You can test it on the test suite distributed with the project or you can try the compiler with your own programs in `VYP08` language. The task was accomplished.

There is no doubt that `VYP08` language is completely useless. The goal of this project is not to define a new revolutionary programming language. Instead of that, the work shows benefits of Flex/Bison usage for development of compilers like `vyp08`.

In chapter 3 are presented generic and reusable compiler modules as the main goal of the project. This main goal has been reached and these modules can be used (with minimal modifications) in development of similar compilers. For example the module of scanner has been already used in another project[1], even before the first release of `vyp08`. Source codes of `vyp08` are covered by a free license – hopefully it will help to use these modules somewhere.

[1] The scanner module was used in a robot simulator which is coming soon...

Bibliography

[1] G. Colvin, B. Dawes, and D. Adler. Boost Smart Pointers. http://www.boost.org/doc/libs/1_35_0/libs/smart_ptr/smart_ptr.htm, 2002.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1997.

[3] Lan Gao. FLEX Tutorial. http://www.cs.ucr.edu/~lgao/teaching/flex.html, 2008.

[4] A. Meduna and R. Lukáš. VYP – Bottom-Up Parsing. https://www.fit.vutbr.cz/study/courses/VYP/private/prednesy/Vyp06-en.pdf, 2008.

[5] R. Pecinovský. *Návrhové vzory*. 2007.

[6] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, special edition edition, 1997.

[7] H. Sutter and A. Alexandrescu. *C++ 101 programovacích technik*. Zoner Press, 2005.

[8] The Flex Project. Generating C++ Scanners. http://flex.sourceforge.net/manual/Cxx.html, 2007.

[9] M. Češka, T. Vojnar, and A. Smrčka. Teoretická informatika - studijní opora. https://www.fit.vutbr.cz/study/courses/TIN/public/Texty/oporaTIN.pdf, 2007.