Implementing a VSOP Compiler

François Rozet

May 8, 2020

1 Modules

The implemented compiler is composed of three main components : a lexer that uses the `Flex` framework for scanning tokens, a parser built with `Bison` that produces an *abstract syntax tree* and, finally, the AST itself, that performs type-checking and code generation at the same time using the `LLVM C++` API.

All the binding and auxiliary code is written in standard `C++14`.

1.1 Lexer - `vsop.lex`

The implemented regular expressions are strongly inspired from the VSOP manual, with some exceptions.

For instance, string literals are not matched with a single regular expression but with a *sub-scanner*. This scanner is entered when opening double-quotes (`\"`) are encountered and isn't left until closing double-quotes are read. This allows, for example, to store spaces, tabulations, etc. while they are ignored in the main scanner.

Multi-line comments are treated similarly, with the addition of a *stack* to remember the locations of unmatched `(*`.

Also, to reduce the number of rules, keywords were not added in the regular expression set. Instead, they are matched with the same expression as object identifiers. Then, when a string is matched, it is checked whether it is a keyword or not using a *keyword table*.

1.1.1 Adhoc regular expression matching – `adhoc/`

Actually, the `Flex` lexer wasn't the first lexer implemented. As a first attempt, a small *regular expression* `C++` library was implemented.

The main goal was to leverage the *operator overloading* capability of `C++` to be able to write directly EBNF-like regular expressions.

Eventually, it was possible to write rules such as

```
underscore = equality("_");
lowercase_letter = range("a", "z");
uppercase_letter = range("A", "Z");
digit = range("0", "9");
letter = lowercase_letter | uppercase_letter;
object_identifier = lowercase_letter + (letter | digit |
   underscore)++;
```

directly in C++.

Even thought it worked quite well, this method was abandoned in favor of Flex mostly because of its integration with Bison.

Also, one should mention that the efficiency of the method was probably terrible as parallel matching expressions (|) were never "merged" together, thus producing an *non-deterministic automaton*.

Note. The sources files are located in the adhoc/ sub-folder. The vsopc executable produced by the Makefile only supports -lex.

1.2 Parser - vsop.y

Bison generates, by default, a *bottom-up* LALR[1] parser. This is especially appropriate for building an AST as it is easier to do it *leaf-to-root* than the inverse.

For project 2, the required representation fitted quite well with the idea of a per-node context-free function that would return its own representation, while recursively asking for the representation of its children, if any.

Therefore, the choice was made to implement this function as a virtual *class member* method named toString, leveraging the inheritance capabilities of C++. However, for this to work, it is necessary that each node holds *references* (pointers) to its children nodes instead of the actual objects. If not done carefully, this could very easily lead to memory leaks. Hopefully, C++14 has *smart pointers* that automatically manage their memory space and will deallocate it when they go out of scope.

1.2.1 Error detection and recovery

By default, Bison stops at soon as it reaches an error. Although it would have met the requirements for the project, is is preferable to continue parsing to report more errors.

Fortunately, Bison possesses a special token error that allows to introduce specific rules to recover from errors. Very roughly, error can force the situation to fit a rule, by discarding part of the semantic context, *i.e.* part of the stack, and part of the input.

The way it was used in this project is similar to *panic-mode* recovery, *i.e.* skipping until the next synchronizing token. For instance, for block expressions,

[1] Look-Ahead Left-to-right Rightmost derivation

```
block     = "{" block-aux ;
block-aux = expr  "}" | expr  ";" block-aux
          | error "}" | error ";" block-aux
          | error block block-aux ;
```

As one can see, `error` is here used to recover from any wrongly formatted `expr`.

However, the `error` token is also dangerous to use as it can skip any token. Especially, it can skip an opening delimiter such as `{` and stop at its counterpart, leaving another `}` dangling somewhere further. The last line of the above grammar prevent this.

Another error detection mechanism that was implemented concerns typos. In VSPO, type identifiers must start with an uppercase letter while object identifiers must start with a lowercase letter. Therefore, for just one minor typo, the entire parsing process could be altered. To prevent this to happen, a rule has been added so that misplaced type identifiers can be *converted* to object identifiers, if needed. The inverse was also implemented.

For example,

```
class Table { NumberOfLegs : int32 <- 4; }
class Dog extends table {}
```

would produce the following error messages :

```
syntax error, unexpected type-identifier NumberOfLegs,
   replaced by myNumberOfLegs
syntax error, unexpected object-identifier table,
   replaced by Table
```

1.3 AST - `ast.hpp/cpp`

For type-checking and code generation, it was chosen to pursue with a simple object-oriented approach : each node would determine their own type and code, based on the set of available classes and methods.

It was decided to decompose the process in two passes : a declaration pass and a code generation pass.

1.3.1 Declaration

The declaration pass is responsible for detecting and fixing

1. cyclic class definitions;
2. redefinitions of classes, methods, fields and formals;
3. overwriting of fields;
4. overwriting of methods with different different signatures;
5. invalid (unknown) field types;
6. invalid method signatures.

To detect inheritance cycles, a table of classes was built. Initially, this table only contains the `Object` class. Then, classes are processed iteratively. A class is added to the table only if a its parent is already in. If one or more classes are added to the table in a pass, a new pass is performed.

The algorithm stops when none of the remaining classes have their parent in the table. Therefore, they either have an ill-formed (parent is unknown) or cyclic inheritance scheme. Redefinition and overwriting checks are also performed using table of methods, fields or formals.

The other task of the declaration pass is to *declare* the class structures, virtual tables and method prototypes. This was done primarily using the `LLVM C++` API. Most functions of this API[2] requires either a *context*, a *builder* or a *module*. To ease the sharing of these three components, an `LLVMHelper` class was implemented (cf. `llvm.hpp`).

Note. Even though the declaration pass is done recursively, the deepest it goes in the abstract syntax tree are formals.

1.3.2 Code generation

The code generation pass is pretty straight forward, yet it was the longest to implement as each class of the AST required a different class method. The hardest part was probably to keep the *scope*, *i.e.* the set of known identifiers and their binding, consistent through the tree traversal.

To do so, a *named values manager* was added to the `LLVMHelper` class. It allows to seamlessly allocate, store and load from stack memory without worrying about erasing previously declared values. It basically is a table of stacks : each name is associated to a stack of values it was declared with, in the innermost declaration order. Eventually, declaring a new variable comes down to pushing a new entry at the top of the stack while losing the reference of a variable (like at the end of a method) comes down to popping the top entry.

It should be mentionned that the type `unit` was interpreted like the type `void` in `C`, *i.e.* the absence of value. Therefore, fields and formals with such type are not present in the declaration of respectively structures and methods. However, they are added to the named values manager (with no corresponding value) so that they can be referenced later on. This can be viewed as a single-step *constant propagation*.

2 Extensions

In its current form, the VSOP language is not very convenient to program with. Therefore, several extensions have been implemented in order to make it a bit more enjoyable and useful to play with.

[2]It was exceptionally painful to go trough the API "documentation" as it is barely documented. Yet, it was worth it as the whole program is represented in terms of API classes which are quite modular. Eventually, not a single line of `LLVM` assembly was written by hand during this project.

It should be noted that switching to the extended mode (`-ext`) doesn't modify the AST at all. In fact, only the lexer and parser are modified, and not by much : a the beginning of the scanning process, the lexer reads a global variable named `yymode` that determines in what mode it should enter. There are four modes : `START_LEXER`, `START_EXT_LEXER`, `START_PARSER` and `START_EXT_PARSER`.

For `EXT` modes, it will use an extended list of keywords and operators. It will also send the `yymode` as a the first token to the parser such that this one can modify its *starting state*. For example, when it receives `START_LEXER` (or `START_EXT_LEXER`) the parser will not build an AST. Instead, it will simply print the tokens (with the required convention) on standard output.

2.1 Floating point arithmetic

As suggested in the manual, having only integers (`int32`) as numeric type is quite limiting. Thus, along with the keyword `double`, floating point arithmetic operations (`"+"`, `"-"`, `"*"`, etc.) have been added to the language.

It was also decided that both numeric types (`int32` and `double`) would always *agree*, meaning that one can be used where the other is required and inversely. This required the implementation of *implicit* casting.

Finally, for operations mixing `int32` and `double`, it was chosen to always cast to `double` as it is the most expressive ($\mathbb{Z} \subset \mathbb{R}$).

A rule for parsing floating point literals was also added[3] to the lexer :

```
real-literal = digit { digit } "." {digit}
             | { digit } "." digit { digit } ;
```

2.2 Control-flow

The `for` control-flow structure was added to VSOP as

```
for = "for" object-identifier "<-" expr "to" expr "do"
   expr ;
```

where the first and second `expr` are `int32` and are respectively the lower and upper bound (included) of the iterating range[4].

Also, in VSOP there is no way to exit prematurely such loop. Therefore, the `break` keyword was added. It also works for `while` loops.

2.3 Lets-In construct

In most programs, multiple new variables have to be declared. As said in the manual, this is quite painful to do with the *let-in* construct.

[3] In fact, this rule is "silenced" in the `vsop.lex` file to keep the lexer consistent with the statement when not in `-ext` mode.

[4] The bounding values are computed only once, at the start of the loop and cannot be modified later.

The *lets-in* construct was introduced as a generalized *let-in* construct.

```
lets = "lets" "(" field { "," field } ")" "in" expr ;
```

2.4 Missing operators

Some missing common operators where added : `"mod"`, `"or"`, `">"`, `">="` and `"!="`.

Note. Equivalently to `"and"`, `"or"` is *short-circuiting*.

2.5 Top-level functions

The possibility to define *top-level functions*, *i.e.* not linked to any class, was added to the language.

```
program = class | method { class | method } ;
```

Furthermore, it isn't required anymore to define a class `Main` with a method `main` : a top-level function `main` with the right signature will do the trick.

Also the method calling convention was modified a little bit : if *no scope is provided*, VSOP will give priority to top-level functions, meaning that if an ambiguity between a top-level function and a class method arise, *i.e.* if they have the same identifier, the function will be chosen. However, specifying a scope, including `"self"`, prevents from calling a top-level function.

2.6 Foreign function interface (FFI)

With the implementation of top-level functions, it was pretty natural to include the possibility of calling *foreign functions* as well. As suggested in the manual, the keyword `"extern"` was added to declare the prototype of foreign functions.

```
prototype = object-identifier "(" formals ")" ":" type ;
method    = prototype block
          | "extern" prototype
          | "extern" "vararg" prototype ;
```

As one can see, external declarations are not bounded to top-level functions. Indeed, one could declare the prototype of a class method and link its definition (even in another language) at the *linking* stage. With some further work, like the creation of *header* files, this could easily lead to *separate compilation*.

Also, some foreign functions, like `printf`, have a *variable* number of arguments. Those are said to be *variadic* functions. Therefore the `"vararg"`[5] keyword was added so that such function could be declared and called within VSOP. It is however not possible to *define* a variadic function/method within VSOP.

In terms of AST textual representation, the declaration

[5]Commonly, the ellipsis symbol `"..."` is placed as the last formal to symbolize that a function is variadic. However, it wasn't easy to implement in VSOP without changing too much the grammar.

```
extern vararg printf ( s : string ) : int32 ;
```

would look like

```
Method(printf, [Formal(s, string)]..., int32)
```

2.7 Optimization passes

Some optimization passes were applied to the created functions using the `FunctionPassManager` of the `LLVM C++` API.

For example, `CFGSimplificationPass` simplifies the control flow graph, deleting unreachable blocks or merging successive single-successor/predecessor blocks.

2.8 Test files

Some extended test files (`.vsopx`) are provided in the `tests` folder.

3 Limitations and retrospective

Even tough it is presented in the extension section of the manual, the *garbage collector* should be a key feature of VSOP. Unfortunately, given that I made this project alone, I didn't have the time to implement it. However, it shouldn't be too hard as the `LLVM C++` API has some builtin procedure to add a GC to an existing module. Also, as said earlier, with a bit more work I could have implemented separate compilation.

Another key feature of most imperative programming languages are arrays, that allow fast(er) data storage and access.

Retrospectively, the choice that cost me the most time was probably the *simple oriented-object* approach. Creating a *visitor* could have helped to encapsulate better the different passes and to have more code shared among nodes.

Also, the `LLVMHelper` class was actually created quite late (even after some extensions). It would have been **much** easier to implement the code generation with this helper available from the start.

However, a choice I don't regret at all is the one to implement my compiler using `C++`, even-though it as drawbacks; the greatest of all being that is very hard to test small part of the code separately before all the files compile.

4 Feedback

As said earlier, I chose to realize this project completely alone, even though it was strongly advised to do it by group of two. I don't regret that choice as I tried a lot of different techniques, merging and rebasing a lot of commits, which is not very compatible with groups for a project of such "small" scale.

As a consequence I spent a **lot** of time on this assignment. I estimate it to be between 120 h and 150 h – I couldn't estimate the time precisely because of the rebase. However, all this time spent was pretty enjoyable as I learned quite a lot during my web searches. In fact, I believe that I will remember much longer what I learned in the project than in the theoretical course, even though the latter helped me quite a lot as they were strongly linked, which is a good point !

4.1 VSOP

In terms of the VSOP language itself, I think it could be interesting to include floating point arithmetic from the start as it is harder to include it afterwards as an extension.

Also, the special object identifier `"self"` is more of a keyword to me, as there are as many places where it cannot be used as places where it can. Furthermore, considering it as a keyword allows to detect invalid assignations (or declarations) to `"self"` earlier, *i.e.* in the syntax analysis.

I also find the use of `unit` as a type for fields and formals quite disturbing, and probably not very useful. The only usage I can see for fields is printing messages at initialization. Instead, the implementation of an explicit class *constructor* could be used.

Finally, I think a method with the return type `unit` shouldn't necessarily require its `block` to end by a `unit` value, similarly to `while` loops.