A Domain-Specific Language Implementation Framework for C++ Based on S-expressions

Aleksandar Stojanović*, Silvio Plehati†, Željko Kovačević‡
Zagreb University of Applied Sciences, Zagreb, Croatia
aleksandar.stojanovic@tvz.hr* silvio.plehati@tvz.hr† zeljko.kovacevic@tvz.hr‡

***Abstract*—Domain-specific languages (DSLs) are languages designed and implemented for a specific application domain. Designing and implementing such languages is hard because they require both domain-specific and language implementation knowledge. However, they offer its users substantial gains in productivity because they consist of constructs that directly represent domain-specific concepts.**

The absence of extensible and easy-to-use domain-specific language frameworks for C++ has made it difficult for developers to build such languages. Designing syntax, implementing parser, the interpreter and its runtime environment is time consuming, error prone and generally requires lots of work. In this paper we present a C++ framework for building domain-specific languages with fixed syntax based on S-expressions, but avoiding the strict parenthesization required by them. The framework offers two advantages over implementing a domain-specific language "by hand": 1) the syntax is already built into the framework's scripting language so users do not need to design and implement their own, and 2) the framework provides the execution environment where users just need to implement domain-specific commands by extending the command set of the interpreter. This approach not only streamlines DSL development but also demonstrates versatility across various domains, as shown by our examples.

Keywords—domain-specific language, framework, scripting, S-expressions, C++

I. Introduction

Domain-specific languages (DSLs) are languages designed for specific purpose [1]. They include a wide variety of languages: markup languages like HTML, data exchange languages like XML and JSON, database query languages like SQL, programming languages like R and Prolog, syntax specification languages like EBNF and many others. Compared to general-purpose programming languages, the benefits of domain-specific languages are many: they are more expressive for their domain of usage, they are easier to use for domain specialists and, consequently, users of such languages are more productive. A good example is SQL: querying a database is much more straightforward with a language that supports relational data model directly than searching and joining tables in a general-purpose programming language like C or Python.

Implementing a DSL from scratch, especially a domain-specific programming language, can be time consuming and error-prone without special knowledge and skills in language implementation. A DSL can be implemented in various ways, depending on the application domain and requirements. Most common are [1]: interpreter, compiler, preprocessor, embedding, extensible compiler or interpreter, commercial tools, and hybrid approach. In this paper we focus on extensible interpreter approach. We chose this approach for the following reasons: 1) to allow users of the framework to create DSLs by extending the interpreter with new capabilities, using a fast general-purpose programming language (C++), 2) to allow users of DSLs created with the framework to use them as stand-alone languages, ensuring a broad and flexible application scope, and 3) to enable integration of DSLs created with the framework into C++ programs.

In this paper we introduce an extensible C++ framework for implementing DSLs similar to typical dynamic scripting languages. The basic idea behind the framework is to allow users to create such a DSL without having to implement it from scratch, but still get reasonable flexibility and performance. To achieve this, the framework provides the following:

- A scripting language that serves as the basis for DSLs. The syntax of the language is based on S-expressions to make its use as simple and as flexible as possible where every construct has the same syntactical structure.
- The API to the interpreter and the runtime environment that allows users to implement a wide variety of extensions, from simple commands like arithmetic operations to commands for flow control, conditional evaluation, pattern matching and others.

Although other programming languages (like Python) can also be extended in C/C++, they only support extensions at the library level, not the core language level. For example, adding a *switch*-like statement to Python would require modifications to its grammar, parser and possibly the runtime system. In contrast, our framework is designed to allow extensions at both, the core language and the library levels (see section III).

At the time of this writing we are not aware of any similar frameworks for C++. There are some more recent tools like PowerShell that provide somewhat similar capabilities for .NET, like adding new commands through the base class and embedding scripts in a .NET program.

The rest of the paper is organized as follows. Section II provides an overview of related work in this area. Section III describes the framework and its scripting language in more detail. Section IV provides a few short examples of

simple domain-specific languages defined using the framework. Section V discusses results from measuring performance of the framework's scripting language execution and compares it to some other similar languages. Finally, section VI completes the paper with the conclusion.

II. Related Work

An excellent overview of various DSL aspects is given in [1]–[3]. They cover basic principles of how to design, develop and implement a DSL. Implementation approaches are covered in detail in textbooks like [4]–[10]. Specifying the syntax of a language, developing the parser, interpreting or compiling the source code are demonstrated through numerous examples.

There are a number of tools available for implementing domain-specific languages in C++, like ANTLR [11], YACC [12], and Boost.Spirit [13]. They could be roughly divided into two categories: preprocessors and parser generators. The former is often based on *template metaprogramming* [14], [15].

A. Preprocessors

In [16] a C++ compiler extension is introduced which enables the integration of domain-specific languages directly within a C++ codebase, eliminating the necessity to segregate the DSL source code from the C++ code. In [17] the authors discuss how domain-specific languages embedded in C++ enable users to create domain-specific code that is both easy to write and safe, while maintaining high performance. This approach provides access to the extensive low-level capabilities of C and C++, along with the vast array of libraries in the C/C++ ecosystem. The paper [18] introduces an embedded domain-specific language in C++ for representing stream parallelism, utilizing standard C++11 attribute annotations. In [19] a framework for building domain-specific languages in C++ using type-based multi-stage programming is proposed. In [20] an extension layer for C++ is described that supports multi-stage generative metaprogramming. It allows programmers to write compile-time metaprograms using the same language constructs as normal programs, and to share design and code reuse between them.

Each of these approaches utilizes C++ as the core component for their DSL, either by translating the DSL into C++ or by augmenting C++ with specific features to support the DSL. While these approaches yield higher performance compared to our framework, they also introduce a significant dependency on C++. This strong coupling necessitates that users possess a certain level of C++ expertise to effectively engage with the DSL. In contrast, our framework is designed to enable users to use DSLs without being tied to any particular programming language, eliminating the need for C++ knowledge among end users. The prerequisite for C++ expertise is confined exclusively to those who are developing DSLs within our framework, thereby broadening accessibility and simplifying the user experience.

B. Parser Generators

Parser generators exist since the early days of Unix. YACC (Yet Another Compiler-Compiler) [12] is a tool that generates parsers in C code, based on LALR grammar as input. Another similar, but more modern and capable tool, is ANTLR (ANother Tool for Language Recognition) [11]. It can generate parsers in many languages based on LL(k) grammars.

Compared to our framework, parser generators represent a more limited solution for DSL implementation. While they efficiently create parsers based on the language's grammar, they fall short by not supplying essential components required to execute DSL source code, such as the interpreter and the runtime environment. This partial approach contrasts with our framework's holistic strategy, which encompasses the full spectrum of DSL execution needs.

III. The Framework

The main purpose of our framework is to allow users to define new *commands* in the framework and access those commands from the built-in scripting language. The goal is to provide as much flexibility for DSL creators as possible, without them having to know the details of the framework itself. Users implement their DSLs by defining new domain-specific commands as *class*es or *struct*s that inherit from a base class provided by the framework. The base class contains the API to the interpreter and the runtime environment to provide access to various facilities such as symbol tables, the abstract syntax tree nodes and evaluation of source code fragments. After adding a new command to the runtime environment, the command becomes available through the framework's scripting language. This is illustrated in Fig. 1.

New commands can be added in two ways, by either 1) defining new subprograms (functions) in the scripting language itself, or 2) implementing new commands in C++ at the framework level. The first approach is simpler for the DSL user, but suffers from several drawbacks:

- Since the scripting language is interpreted, it may lack acceptable performance (for applications where that's important).
- Certain capabilities are not provided in the scripting language (like low-level access to various computer resources, specific algorithms and/or libraries, usage of advanced programming facilities like threads, processes, GPU processing, etc.).
- Defining commands with domain-specific control flow is not (currently) possible. For example, implementing a selection statement similar to *if* is not possible without access to the framework's API.

For these reasons, the framework is intentionally designed to align with approach (2). However, this specific approach has its own drawbacks: it necessitates a comprehensive understanding of C++, the framework's API, and some insight into its internal mechanisms.

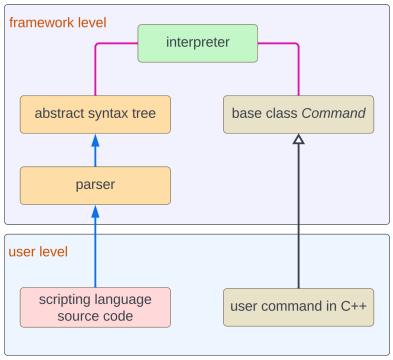

Fig. 1: Architecture of the DSL implementation framework for C++.

Listing 1: The main part of the EBNF for the scripting language (productions for *symbol*, *string*, *double*, *int* and *bool* are not shown).

```
1 block     = statement, {statement};
2 statement = command,("|", statement | ";");
3 command   = parameter, {parameter};
4 parameter = "{", block, "}" | symbol
5             | string | double | int | bool;
```

A. The Framework's Scripting Language

The framework's scripting language serves as user interface to user-defined commands. The fundamental building block of the scripting language is the command. Technically, commands are functions that can take zero or more parameters and return a result. The basic form of a command is

$$eval : \{f\ p_1\ p_2\ ...\ p_n\} \rightarrow result \qquad (1)$$

where f is either a command name or a command object and p_k is command's parameter (parameters are optional). This forms an expression. Both f and p_k can either be a constant or an expression.

The syntax in (1) is basically the S-expression notation [21], but the framework supports some shortcuts to minimize the number of curly braces: statements at the top level are automatically placed inside curly braces and comma can be used to avoid having to write curly braces in some expression forms, like `if, > n 0 {return -n}` instead of `if {> n 0} {return -n}`. There is no fixed semantics of the language because it depends on how users implement their commands.

The main part of the language's grammar in EBNF [22] is shown in Listing 1. The rules for *symbol*, *string*, *double*, *int* and *bool* are trivial regular expressions and are not shown in the listing. The advantage of this S-expression-based syntax is consistency: every construct has the same structure (including those for flow control, like loops and conditional evaluation). This consistent syntactical structure makes it easier to define new commands in the framework and, more importantly, allows the framework's scripting language interpreter to combine those commands with other commands into more complex expressions.

Listing 2 shows a function for calculating the absolute value of a number, written in the framework's scripting language. The syntactical structure of defining the function *abs* is shown in Fig. 2. Note that a block of code can also

Listing 2: Function *abs* in the framework's scripting language.

```
1 fn abs n {
2   if {< n 0} {
3     return -n
4   } else {
5     return n
6   }
7 }
```

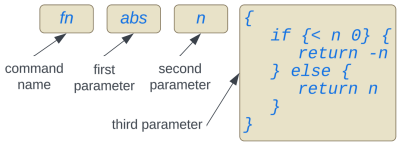

Fig. 2: Syntactical structure of command *fn*.

be an argument for a command parameter as shown for the *fn* and *if* commands in listing 2 (see Fig. 2).

B. Defining New Commands

Commands like *fn, if, return* and many others are already defined in the framework to make the scripting language usable out of the box. Those commands are defined in the same way users would define their domain-specific or any other commands.

Suppose we want to add a new command for calculating the power of a number, call it *pow*, and use it like `pow <base> <exponent>`. Listing 3 shows how a user would implement such a command in our framework. This command definition consists of the following: name ("pow", line 2), return type NUMBER (line 2), parameters with their name and type (line 3), description (line 4), and command functionality (function *exec* in line 6). The name of the command is what users will type in the scripting language to invoke the command. The return type is necessary for the runtime to check in cases where types need to match (when the result of one command is an input to another). In the listing, the return type is NUMBER, which means the function returns either an integer or a double (the type information is stored in the *Value* object). The *Params* object specifies the name and type of parameters. Parameters can be accessed by that name from the *exec* function, as seen in lines 7..8. The description is used by another command that is part of the framework to print useful information about a command (like parameter information, return type and description). The function *exec* is the implementation of the command - what it does and what it returns as result.

After the new command object (the instance of class

Listing 3: Command for the *power* function.

```
1  struct Power : Command {
2      Power() : Command{"pow", NUMBER, Params(this,
3                    M(NUMBER, "base"), M(NUMBER, "exp")),
4          "Returns <base> to the power of <exp>."} {}
5
6      void exec(Value* r, Params& p) override {
7          double n = pow(p["base"]->double_value,
8                         p["exp"]->double_value);
9          set_result(r, n);
10     }
11 };
```

Listing 4: Using the *pow* command for binary-to-decimal conversion.

```
1 fn bin-to-dec a {
2   let n 0; let exp 0
3   for i {- {count a} 1} -1 {  # from high to low
4     let v {* {at a i} {pow 2 exp}}
5     set n {+ n v}; inc exp
6   }
7   return n
8 }
```

Listing 5: Command for the *for*-loop.

```
1  struct For : Command {
2      For() : Command{"for", ANY, Params(this,
3          M(SYMBOL, "ctrl-var"), M(INT, "from"),
4          M(INT, "to"), M(BLOCK, "body")),
5          "The for-loop."} { }
6
7      void exec(Value* r, Params& params) override {
8          Value* var = params["ctrl-var"];
9          Value* range_from = params["from"];
10         Value* range_to = params["to"];
11         Value* body = params["body"];
12         Value& control_var =
13             add_variable(var->text_value,
14                                       range_from);
15         Value body_result;
16         while (control_var.int_value <
17                range_to->int_value) {
18             execute_block(body->block_node,
19                           &body_result);
20             if (return_flag_set()) {
21                 set_result(result,
22                            *get_return_value());
23                 return;
24             }
25             ++(control_var.int_value);
26         }
27         set_result_nil(r);
28     }
29 };
```

Power) is added to the runtime environment, it will become available in the scripting language and usable in combination with other commands. Listing 4 shows a function *bin-to-dec* that converts a binary number given as an array of bits to its decimal representation. The new *pow* command is used in line 4.

As this example shows, users just need to specify what the command's input parameters and result are and the framework handles everything else.

The Listing 5 shows a simplified implementation of command *for* (used for looping). The main API calls are shown in bold. Parameter values are obtained in lines 8..11. The control variable is added to the symbol table in line 12. The looping is implemented by the *while*-loop in lines 16..26. The *execute_block* call executes the body of the *for*-loop the number of times specified by the *from* and *to* parameters. In line 20 a check is made to see if the *return* command was encountered inside the body of the *for*-loop, in which case the execution stops and the result of the *for*-loop is the result of the optional expression specified by command *return*. After all iterations have been performed, the execution stops and NIL is returned as the result of command *for*.

This example demonstrates that the framework is flexible enough to support implementation of more complex commands like flow control, conditional evaluation and others.

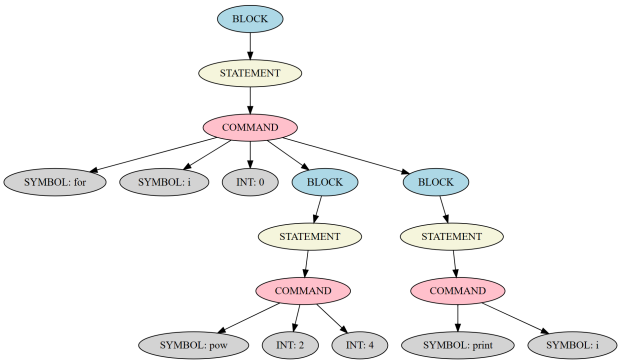

Fig. 3: The abstract syntax tree for expression `{for i 0 {pow 2 4} {print i}}`.

C. The Execution Process

Before execution, the source code is passed to the parser that creates the abstract syntax tree (AST) from it. The AST is then used by the interpreter to evaluate the expressions. To create the AST, the parser does not require any commands to be defined. However, when the interpreter begins the AST evaluation it will look for commands used in the source code, so by that point they must be added to the runtime environment. As an example, for the expression `{for i 0 {pow 2 4} {print i}}` the parser creates the AST like the one shown in Fig. 3.

As the grammar in Listing 1 shows, there can be multiple statements separated by "|" or ";", hence the STATEMENT nodes (the "|" symbol is used for passing the result of one expression to another, similar to pipes in shell languages and the ";" separates multiple expressions on the same line).

The interpreter basically performs left-to-right, depth-first traversal of the AST. In Fig. 3, the leftmost child of the node marked COMMAND is the command name, while the rest of the children are command parameters. After all command parameters are evaluated and assigned to, the interpreter calls the *exec* method on that command to get the result of the (sub)expression. In our example, when the interpreter arives at node "SYMBOL: for" it looks for command with the name "for" in the global symbol table. When it finds it, it evaluates the parameters, verifying that the types match what is specified in the command's constructor. The first and second parameters of command "for" are constants of types SYMBOL and INT, respectively; the framework supports additional types like DOUBLE, NUMBER, BOOL, STRING, ARRAY, DICT, FUNCTION, and BLOCK (the type NUMBER is either INT or DOUBLE) and each *Value* object carries the type information for a value it represents. For the third parameter (the one named "to" in Listing 5) of type INT the interpreter evaluates the subtree at the BLOCK node to get the value for that parameter. The fourth parameter is of type BLOCK so the interpreter does not evaluate that part of the AST but instead assigns that node to the parameter. This parameter is evaluated "manually" later in method *exec* using `execute_block` (line 18) to execute the body of the loop.

Listing 6: DSL for querying CSV files.

```
var result {
    select item price
        from "prices.csv"
        where {> price 100}
}
cout result
```

Listing 7: DSL for generating simple SVG graphics.

```
svg "rect.svg" viewport width 200 height 200 {
  var xc 10; var yc 10; var x 10; var y 2
  for i 0 10 {
    svg rect x xc y yc width 45 height 27
      style "stroke: black; fill: beige;"

    inc xc x; inc yc y; inc x 2; inc y 2
  }
}
```

IV. DSL Examples

In this section we show a few short examples of how a DSL can be designed using this framework. Due to limited space we do not show full implementations, we just show what they look like and briefly describe them. The previous sections showed how new commands can be defined, so here we focus on end user's perspective, i.e. the language aspects.

A. Data Processing DSL

Listing 6 shows a part of a small data access DSL. The command name is *select*. The next two parameters are names of CSV [23] file's columns. The next parameter is symbol *from* followed by a parameter containing the file name. This is followed by another parameter, symbol *where*, followed by a block containing the filtering condition. The symbols *item* and *price* are added to the symbol table where they are assigned values from each row. Those values are checked by the block after the *where* symbol. If the block returns *true* the values for *item* and *price* are included in the result.

B. DSL for Generating SVG

The next example shown in Listing 7 generates an SVG [24] image, shown in Fig. 4, from the script. The command *svg* takes the file name as the first parameter and SVG XML instruction as the second parameter, followed by other parameters, depending on the instruction. The last parameter is the block with the script that generates the SVG file inside the specified viewport. The command *svg rect ...* in line 4 generates SVG instruction like *<rect x="10" y="10" width="45" height="27" style="stroke: black; fill: beige;"/>*.

C. Grammar Specification DSL

As another example, Listing 8 shows a simplified grammar definition for S-expressions in a notation similar to

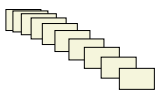

Fig. 4: Image generated by script in Listing 7.

Listing 8: Grammar definition for S-expressions.

```
bnf symbolic-expressions {
  expr     = p-expr ! atom
  p-expr   = "(" operator expr* ")"
  atom     = NUMBER
  operator = SYMBOL ! p-expr
}

# command prompt
> parse symbolic-expressions "(* a (+ b c))"
<BOOL> T
```

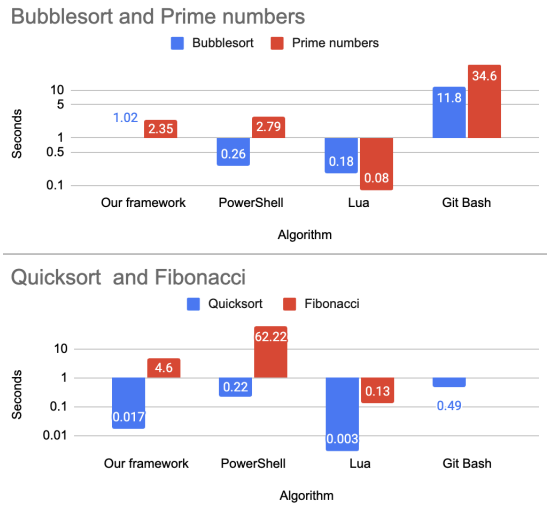

Fig. 5: The execution speed of the framework's interpreter compared to PowerShell, Lua and Git Bash.

BNF. Command *bnf* takes the name of the definition as first parameter and the block with grammar productions as the second (the "!" symbol is used for separating the alternatives). Command *parse* (not defined here, but also part of the DSL) takes the grammar name as the first parameter and a string containing an S-expression and returns *true* (T) if the S-expression is syntactically correct or *false* (F) if it is not.

The examples presented demonstrate the versatility and power of S-expression-based syntax in creating domain-specific languages. The inherent simplicity and uniformity of S-expressions offer a robust foundation for designing DSLs that are both expressive and easy to parse.

V. Results

This section shows performance measurements of the framework's scripting language interpreter compared to three other scripting languages: Powershell [25], Lua [26] and Git Bash [27]. The results are shown in Fig. 5[1]. Athough none of these languages are designed with the same goals as our framework, our intention here is to demonstrate that runtime performance of the framework's scripting language can be comparable to other similar languages.

The algorithms used for measuring performance were Quicksort (1000 elements), 30th Fibonacci number, Bub-

[1]The performance was measured on a computer with Intel i7 processor on 2.7 GHz, 32GB of RAM on a 64-bit Windows 10 operating system and Microsoft C++ compiler v. 19.36.32532 (with optimization for speed).

blesort (2000 elements) and prime numbers in interval 1..100,000 [28], [29]. These algorithms were chosen to measure the performance of function calls through recursion (Quicksort and Fibonacci numbers) and iterative execution (prime numbers and bubble sort). In all four languages these algorithms were implemented in the same way. The number of elements for all algorithms was chosen so that each of these languages can execute them, although in the case of 30th Fibonacci number Git Bash could not finish within an acceptable time period (we recognize that implementing Fibonacci recursively is not practical in terms of performance; however, our objective was to evaluate the efficiency of function calls across these languages).

The results show that our framework has the performance closest to that of PowerShell, significantly better than Git Bash, but also significantly worse than Lua due to differences in purpose and implementation.

Compared to our framework, PowerShell proved to be faster for iterative algorithms, specifically for Bubble sort where it was about four times faster, but significantly slower for recursive algorithms: for Quicksort it was about 15 times slower, similarly for calculating the 30th Fibonacci number. Function calls in PowerShell seem to take significant time, but a deeper analysis of that system was not performed here. This can also be seen in the algorithm for prime numbers where PowerShell is a little slower. Although this algorithm is not implemented recursively, for each number in the given interval a function is called that checks whether it is prime. This is in contrast to the implementation of the Bubble sort algorithm, which consists of two nested *for* loops and no function calls.

VI. Conclusion

In this study we introduced a comprehensive C++ framework designed to facilitate the development of DSLs. This framework provides a streamlined method for DSL implementation, eliminating the complexities associated with parser, interpreter, and runtime environment development. Despite its numerous benefits, the framework is not without its limitations. Primarily, it constrains users to the syntax of S-expressions and certain functionalities may prove challenging or unfeasible to implement. For instance, developing a DSL that emulates the logic programming paradigm of Prolog would be problematic due to the framework's inherent orientation towards imperative or functional programming styles. However, as delineated in Section IV, the flexibility afforded by the S-expression-based syntax enables the creation of a diverse array of DSLs, underscoring the framework's potential versatility and utility in the domain of language development. This underscores the framework's value for researchers and practitioners, offering a foundation for future innovations and improvements.

References

[1] M. Mernik, J. Heering, and A. M. Sloane. *When and how to develop domain-specific languages*. *ACM Comput. Surv.*, 37(4), 2005, DOI: 10.1145/1118890.1118892.

[2] L. Shen, X. Chen, R. Liu, H. Wang, and G. Ji. *Domain-specific language techniques for visual computing: a comprehensive study*. *Archives of Computational Methods in Engineering*, 10 2020, DOI: 10.1007/s11831-020-09492-4.

[3] S. Sobernig. *Variability support in DSL development*, pages 33–72. 2020, ISBN: 978-3-030-42151-9.

[4] A. Aho, J. Ullman, R. Sethi, and M. Lam. *Compilers: principles, techniques, and tools*. Addison Wesley, 2nd edition, 2006, ISBN: 978-0321486813.

[5] R. Mak. *Writing compilers and interpreters: a software engineering approach*. Wiley, 3rd edition, 2009, ISBN: 978-0470177075.

[6] R. W. Sebesta. *Concepts of programming languages*. Pearson, 11th edition, 2016, ISBN: 1-292-10055-9.

[7] R. Nystrom. *Crafting interpreters*. Genever Benning, 11th edition, 2021, ISBN: 0990582930.

[8] A. Ranta. *Implementing programming languages*. College Publications, 2012, ISBN: 1848900643.

[9] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of programming languages*. MIT Press, 2008, ISBN: 978-0-262-06279-4.

[10] M. Boersma. *Building user-friendly DSLs*. Manning, 2024, ISBN: 1617296473.

[11] T. Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2nd edition, 2013, ISBN: 1934356999.

[12] S. C. Johnson. *YACC: Yet Another Compiler-Compiler*. Technical report, ATT Bell Laboratories, 1975.

[13] boost.org. *Boost.Spirit*. https://www.boost.org/doc/libs/1_78_0/libs/spirit/doc/html/index.html, 2011. Accessed: 2024-03-14.

[14] D. Abrahams and A. Gurtovoy. *C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond*. 2004, ISBN: 0321227255.

[15] D. Vandevoorde and N. M. Josuttis. *C++ templates: the complete guide*. Addison-Wesley, 2017, ISBN: 0321714121.

[16] H. Finkel, A. McCaskey, T. Popoola, D. Lyakh, and J. Doerfert. *Really embedding domain-specific languages into C++*, 2020, DOI: 10.1109/LLVMHPCHiPar51896.2020.00012.

[17] S. T. Kozacik, E. M. Chao, A. L. Paolini, J. Bonnett, and E. J. Kelmelis. *Improving developer productivity with C++ embedded domain specific languages*. In *Defense + Security*, 2017, DOI: 10.1117/12.2264800.

[18] D. Griebler, M. Danelutto, M. Torquati, and L. G. Fernandes. *An embedded C++ domain-specific language for stream parallelism*. In *International Conference on Parallel Computing*, 2015, DOI: 10.3233/978-1-61499-621-7-317.

[19] A. Brahmakshatriya and S. Amarasinghe. *BuildIt: A type-based multi-stage programming framework for code generation in C++*. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 39–51, 2021, DOI: 10.1109/CGO51591.2021.9370333.

[20] Y. Lilis and A. Savidis. *Meta C++: An extension layer for multi-stage generative metaprogramming*. *The Journal of Object Technology*, 18:1:1, 03 2019, DOI: 10.5381/jot.2019.18.1.a1.

[21] J. McCarthy. *Recursive functions of symbolic expressions and their computation by machine, part I*. *Communications of the ACM*, 3(4):184–195, 1960, DOI: https://doi.org/10.1145/367177.367199.

[22] ISO/IEC. *ISO/IEC 14977:1996: Information technology - syntactic metalanguage - extended BNF*. [1](https://www.bibme.org/bibtex), 2008. Accessed: 2024-01-11.

[23] Yakov Shafranovich. *Common format and MIME type for comma-separated values (CSV) files*. RFC 4180, October 2005. Accessed: 2024-01-11.

[24] World Wide Web Consortium. *Scalable Vector Graphics (SVG) 1.1 (second edition)*. https://www.w3.org/TR/SVG/, 2011. W3C Recommendation.

[25] L. Holmes. *PowerShell cookbook*. O'Reilly, 4th edition, 2021, ISBN: 109810160X.

[26] R. Ierusalimschy. *Programming in Lua*. Lua.Org, 4th edition, 2016, ISBN: 8590379868.

[27] C. Newham. *Learning the Bash shell*. O'Reilly, 3rd edition, 2005, ISBN: 0596009658.

[28] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. The MIT Press, Cambridge, Massachusetts, 3rd edition, 2009, ISBN: 978-0262033848.

[29] M. T. Goodrich, R. Tamassia, and M. H. Goldwasser. *Data structures and algorithms in Python*. Wiley, 2013, ISBN: 978-1-118-26844-2.