Storm: A Language Platform for Interacting and Extensible Languages (Tool Demo)

Filip Strömbäck

The self-archived postprint version of this journal article is available at Linköping University Institutional Repository (DiVA):
http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-154765

N.B.: When citing this work, cite the original publication.
Strömbäck, F., (2018), Storm: A Language Platform for Interacting and Extensible Languages (Tool Demo), *PROCEEDINGS OF THE 11TH ACM SIGPLAN INTERNATIONAL CONFERENCE ON SOFTWARE LANGUAGE ENGINEERING (SLE 18)*, , 60-64.
https://doi.org/10.1145/3276604.3276982

Original publication available at:
https://doi.org/10.1145/3276604.3276982

Copyright: Association for Computing Machinery (ACM)
http://www.acm.org/
© ACM 2019. This is the author's version of the work. It is posted here for your personal use. Not for redistribution.

LINKÖPING UNIVERSITY

Storm: A Language Platform for Interacting and Extensible Languages (Tool Demo)

Filip Strömbäck
Department of Computer and Information Science
Linköping University
Linköping, Sweden
filip.stromback@liu.se

Abstract

The ability to extend programming languages with domain-specific concepts is becoming an essential technology for developing complex software. However, many domain-specific languages are implemented in a way that interact poorly with the host language. There are a number of tools that aim to improve the situation by simplifying the creation of domain-specific languages, and allow easier interactions between the host language and the domain-specific language. However, many of these tools are limited to a single host language, and rarely allow extending the language used for language creation. To improve the situation, we created the language platform Storm, which aims to make the creation and usage of multiple extensible languages easy and seamless. This is accomplished by means of a shared, standardized namespace and in-process code generation, which gives Storm a high degree of extensibility, making it possible to extend or replace the built-in languages at will.

CCS Concepts • **Software and its engineering → Extensible languages**; *Domain specific languages.*

Keywords domain-specific language, DSL, extensible language

ACM Reference Format:
Filip Strömbäck. 2018. Storm: A Language Platform for Interacting and Extensible Languages (Tool Demo). In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3276604.3276982

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SLE '18, November 5–6, 2018, Boston, MA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6029-6/18/11…$15.00
https://doi.org/10.1145/3276604.3276982

1 Background

Domain-specific languages (DSLs) are becoming increasingly popular in many situations [4]. For example, many popular GUI libraries, such as Gtk+, Android, and WPF include some kind of DSL that allow defining layout more conveniently than in a general purpose programming language. These frameworks clearly demonstrate the usefulness of DSLs, but they have some shortcomings. Most notably, since the host language and the DSL are separate, interactions between the two are usually not straightforward. For example, to interact with elements defined in the layout, it is necessary to manually retrieve them using a numeric ID provided in the form of a constant. Furthermore, this means that the host language is unaware of the type of the component being accessed, and thus it is up to the programmer to provide the type information by using explicit casts.

These issues could be addressed by using a tool like Xtext [2], Spoofax [8] or MontiCore [5] to develop the layout language. DSLs developed using these tools are implemented in terms of a grammar with a program describing how to compile the DSL into a host language (Java in this case). Since the DSL compiles to the host language, the two languages may interact to a higher degree than in the case of the syntax language in Android.

Aside from the previously mentioned *external* DSLs, it is often desirable to *embed* a DSL inside a host language to simplify some aspect of the current domain. Since the DSL is embedded, such DSLs are usually able to interact with the host language to a higher degree than external DSLs. Furthermore, creating an embedded DSL is often faster than creating an external DSL, as much functionality from the host language can be easily reused [6]. As with external DSLs, there are a multitude of tools that aid the development of embedded DSLs. One of the earlier examples of allowing syntax extensions in a language is macros in Lisp, which were further refined and brought into focus in Racket with pattern-based macros and other facilities that ease the creation of DSLs [3, 11]. This approach provides a high degree of flexibility as arbitrary code may be executed during the expansion of a macro. Macros are also easily composable, which allows using multiple extensions simultaneously. However, this approach requires that the DSLs are represented as S-expressions unless a custom *reader* is used [3],

which parses the source text into a suitable representation. Since the implementation of a reader is up to the language designer, they are not easily composable.

There are a number of tools, such as SugarJ [1], ableJ [12] and ableC [7], that provide composable syntax extensions to other languages (Java and C in this example). In contrast to Xtext, Spoofax and MontiCore, these tools focus on embedded DSLs which are typically smaller than external DSLs. Therefore, these tools make it convenient to select which extensions to use on a file-by-file basis. Extensions are, much like with Xtext, Spoofax and MontiCore, implemented in terms of a grammar that augments the host language's grammar and a description of the semantics in terms of the host language. While this approach allows a greater flexibility in the syntax of the embedded language compared to Racket, it has drawbacks. First and foremost, a language extension is limited by the capabilities of the host language, making it difficult, if not impossible, to implement a DSL that lowers the level of abstraction or bypasses the type system, for example `unsafe` blocks in Rust[1]. Furthermore, since the host language is compiled and executed as separate steps, it is generally not possible to extend the syntax or semantics of the language used to describe the language extensions (which is, to some degree, possible in Racket).

In this paper, we present Storm, which is a language platform aiming to address the shortcomings mentioned above. Storm supports creating both embedded and external DSLs that are able to interact, even if the languages are unaware of each other. Storm also provides a runtime environment shared between the compiler and the compiled code, which allows arbitrary code to be executed during compilation. This in turn means that it is possible to extend or replace the languages used for language definitions. Furthermore, since all language implementations are a part of the runtime environment just like any code in the system, a new language may reuse parts of one or more other languages to simplify its implementation.

2 An Overview of Storm

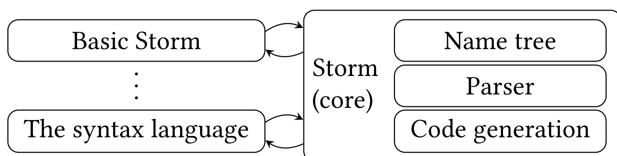

Figure 1. Overview of Storm

As shown in Figure 1, Storm itself is a language agnostic collection of tools that aid development of extensible programming languages [10]. These tools provide standardized interfaces that languages may use to interact with each other, and with the system itself. Most importantly, Storm provides a shared hierarchical namespace called the *name tree*, a parser for context free grammars and in-process code generation. Even though languages are encouraged to comply with the standardized interfaces in order to provide seamless interaction between languages, all interfaces except the runtime system and the code generation are optional, which means that a language is free to ignore interfaces that do not suit its semantics.

Languages in Storm are, as can be seen in Figure 1, implemented as libraries separate from the core system. However, since Storm would be useless without any language, two libraries implementing the languages *Basic Storm* and *the syntax language* are bundled with Storm by default. Even though these languages are bundled with the system, they do not receive special treatment and could have been implemented in another language. Since all languages are treated equally by Storm, it is possible to use any language that conforms to the relevant interfaces to create new languages and language extensions. For example, as we shall see, it is possible to create a language extension to Basic Storm that simplifies the implementation of future language extensions.

In order to make this kind of extensibility possible, Storm does not distinguish between compilation and execution of the compiled program; code is generated inside the compiler process and then executed there, meaning it is possible to execute arbitrary code in any language during compilation. Furthermore, Storm employs *lazy compilation* in most of the system, meaning that the contents of packages and types are not loaded until they are needed and that functions are compiled only when they are executed for the first time. Lazy compilation works in conjunction with the ability to execute arbitrary code to make it possible to implement a language partially in itself, among other things.

Storm is freely available at http://storm-lang.org/.

3 An Extension to Basic Storm

```
1 use lang:bs; use lang:bs:macro;
2
3 Expr unlessExpr(Block b, Expr c, Expr r) {
4   Expr result = pattern(b) {
5     if (!${c})
6       return ${r};
7   }
8   return result;
9 }
```

Listing 1. Using patterns to create an AST.

The example in Listing 1 illustrates the extensibility possible in Storm. The function `unlessExpr` is a function written in Basic Storm that generates an AST for an extension that simplifies early returns in functions. To simplify the implementation, a language extension providing `pattern` blocks

[1]https://doc.rust-lang.org/book/second-edition/ch19-01-unsafe-rust.html

is used. `pattern` blocks are similar to backquotes in Lisp; they evaluate to the AST corresponding to the contents of the block, with some parts possibly provided by expressions that refer to the surrounding context.

In this particular example, the `pattern` block is used to create an AST for an `if`-statement that returns the value of the expression stored in `r` if the condition in `c` is false. These expressions are inserted into the pattern using the `${}` syntax, which allows inserting AST nodes produced by arbitrarily complex Basic Storm expressions.

4 The Syntax Language

The syntax for languages in Storm are often implemented in the syntax language, which is a DSL for describing context-free grammars and their associated *syntax transformations*. The syntax language emits entities in the name tree in a language agnostic representation used by the parser. The representation is not only used for storing the grammar, it also describes the types representing the parse tree produced by the parser. The representation is designed to look like regular types to languages unaware of the syntax language so that all languages may examine and manipulate parse trees as if they were regular data structures. Furthermore, since the representation is language agnostic, it is possible to create other languages for describing syntax for cases where the syntax language is deemed insufficient.

In order to simplify the task of transforming the parse tree into an AST, the syntax language provides a mechanism called *syntax transformations*, which aims to simplify the task by using annotations in the grammar. Each production is associated with a function, the signature of which is specified by the nonterminal at the left-hand side of the production. The function itself examines the node and transforms the node according to annotations in the matched production. This is similar to semantic actions [9], commonly used in parser generators.

```
1  SExpr => TemplateExpr(block, env, create)
2    : "pattern", "(", SExpr(block) env,
3      ")", SExpr @create;
4
5  SAtom => insertExpr(pos, block)
6    : "$" - SPatternExpr @pattern
7    = PatternAtom;
8
9  void SPatternExpr();
10 SPatternExpr : "{", SExpr @expr, "}"
11   = PatternExpr;
```

Listing 2. Parts of the syntax defining `pattern` blocks.

To better understand the syntax language, consider the partial implementation of the syntax for the `pattern` block in Listing 2. Three productions and one nonterminal symbol (called *rule*) are declared. The first production on lines 1-3 declares a production for the `SExpr` rule, which describes expressions in Basic Storm. As such it is declared in the grammar for Basic Storm, and not included in the extension. Thus, the production on lines 1-3 injects new syntax into the language by providing an additional production. Similarly, the production on lines 5-7 augments the `SAtom` rule, which describes the smallest parts of an expression (e.g. literals), with the `${}` syntax. Finally, line 9 declares a new rule, `SPatternExpr`, which is used to describe an expression that produces an AST node which is to be inserted, and lines 10-11 declare a production for that rule (the official release contains additional productions for `SPatternExpr`, which are omitted for brevity).

The example also describes the semantics of the new syntax using syntax transformations. The first two productions in Listing 2 each specify a simple expression (a single function or constructor call) after the `=>` symbol. This expression is evaluated when a corresponding node in the parse tree is evaluated to produce the result of the transformation. The parameters passed to the function on line 1, `block`, `env` and `create` originate either from captured parts of the production (`env` and `create`), or from formal parameters of the rule (`block`). A part of the production is captured by adding a name after the desired part. This causes the corresponding part of the parse tree to be transformed and bound to the specified name. If an `@` is prepended to the name, the node is not transformed before it is bound to the name. This is done for `create`, since the `pattern` block will transform its contents at runtime in another context. As previously mentioned, transformations may also require parameters, which is the case for `SExpr`. `SExpr` is declared as `Expr SExpr(Block block);` which means that the transformation returns an `Expr` instance and requires a `Block` instance as a parameter. Parameters to such rules are supplied where the rule is used in the grammar, as can be seen on line 2.

As previously mentioned, rules and productions accessible to other languages through the name tree, and appear as types to languages unaware of the syntax representation. Each rule appears as an abstract type containing a `transform` method with a signature matching the one in the rule declaration, which is why rule declarations look like C-style function declarations. Each production then appears as a type inheriting from the type of the rule on the left-hand side of the production. This type is anonymous unless a name is specified as on lines 7 and 11. The type overrides the `transform` function with the behavior specified by the syntax transformation, and contain a data member for each captured part of the production. This arrangement allows other languages to easily inspect and modify a parse tree in a convenient manner.

In addition to simplifying interactions with other languages, the syntax language's presence in the name tree is useful for organizing and selecting language extensions.

The parser in Storm maintains a set of packages (i.e. paths into the name tree) specifying the productions that are considered during parsing. Since a rule and a production may be declared separately (which is the case for SExpr and SAtom in Listing 2), this mechanism allows languages to control which languages extensions to use by specifying packages, which allows implementing inclusions similar to SugarJ [1] and ableJ [12]. Furthermore, the associated syntax transformations allow language extensions to inject semantics alongside the syntax without additional mechanisms.

5 Semantics in Basic Storm

The semantics of a language can be implemented in any language in Storm, as long as the language produces functions callable from the syntax language. The goal of the semantics for any language is to eventually produce executable code in the intermediate representation used by Storm. However, this is a low-level representation (similar to x86 assembly), which means that it is often tedious to use directly. Instead, a language may reuse parts of another language implementation for this purpose, which is possible since all language implementations reside in the name tree and are thereby available for other language implementations to use. This approach is used by the syntax language to implement syntax transformations and, as we shall see, the pattern extension.

```
1  class TemplateExpr extends ExprBlock {
2    init(Block b, Expr env, SExpr create) {
3      // ...
4      var atoms = create.allChildren(
5                  named{PatternAtom});
6      for (id, a in atoms) {
7        if (a as PatternAtom) {
8          a.pattern = PatternExprSrc(id);
9        }
10     }
11     // ...
12     ReferSExpr src(create);
13     add(namedExpr(b, SStr("transform"),
14         exprs, Actuals(src)));
15     // ...
16   }
17 }
```

Listing 3. Parts of the semantics for the pattern block.

Listing 3 contains part of the semantics for pattern blocks. A pattern block is implemented in terms of a few Basic Storm expressions wrapped inside a block (blocks are expressions in Basic Storm; they behave like progn in Lisp). As such, the class TemplateExpr, which represents pattern blocks, inherits from ExprBlock. The constructor (lines 2-16) first examines the captured syntax tree, create, to find all expressions that shall be inserted, which are represented by the PatternAtom class (lines 4-5). Each occurrence is then modified by replacing the pattern member with a custom subclass to SPatternExpr that contains information on how to retrieve the actual value to be inserted (lines 6-10). The ASTs inside these nodes are evaluated in another context, and the results are stored for later retrieval (not shown). Finally, the constructor adds a Basic Storm expression to the current block that calls the transform method on a previously created object, which transforms the captured parse tree (lines 13-14). Since Basic Storm does not allow storing arbitrary values as constants in the intermediate representation, a custom AST node, ReferSExpr, which provides this functionality is used on line 13 by generating a small amount of code in the intermediate representation.

6 Conclusion

In this paper, we highlight some of the strengths of the approaches to extensibility used by Storm by examining the implementation of pattern blocks. First and foremost, the example illustrates how the tools and languages provided by Storm can be used to create DSLs that simplify future language creation, which is beyond the capabilities of other similar tools, like SugarJ and ableJ. This is possible since Storm is not tied to a specific language, but rather only defines a handful of interfaces to which languages are expected (but not required) to conform. This combined with the ability to execute arbitrary code during compilation means that a programmer is free to extend or replace the default languages as long as they conform to the relevant parts of Storm's interfaces. This too can be done in any language that supports the relevant interfaces, and does not necessarily have to be done in one of the default languages.

The core structure in Storm is the name tree, which defines the namespace shared between languages. The name tree is not only used for inter-language interactions, Storm also exposes large parts of itself through the name tree. This makes it possible for languages to inspect and modify the running system, which is required in order to generate executable code. Finally, the fact that all language implementations are exposed in the name tree means that other languages are able to reuse parts of these implementations in order to reduce the work required to implement a new language. This ability is used heavily in the implementation of the syntax language; the syntax transformations are implemented by utilizing the functionality of Basic Storm. This was also illustrated in Section 5, where new semantics was implemented in terms of already existing semantics. The example also notes the possibility of providing the missing semantics by generating the required intermediate code directly, even while utilizing other language implementations.

The example presented in this paper is available in the official release at http://storm-lang.org/. The syntax in Listing 2 is located in the file root/lang/bs/macro/syntax.bnf, and

the semantics in Listing 3 is located in the file `root/lang/bs/macro/pattern.bs`.

References

[1] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. SugarJ: Library-based Syntactic Language Extensibility. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 391–406. https://doi.org/10.1145/2048066.2048099

[2] Moritz Eysholdt and Heiko Behrens. 2010. Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA '10)*. ACM, New York, NY, USA, 307–309. https://doi.org/10.1145/1869542.1869625

[3] Matthew Flatt. 2011. Creating Languages in Racket. *Queue* 9, 11, Article 21 (Nov. 2011), 15 pages. https://doi.org/10.1145/2063166.2068896

[4] Debasish Ghosh. 2011. DSL for the Uninitiated. *Queue* 9, 6, Article 10 (June 2011), 12 pages. https://doi.org/10.1145/1989748.1989750

[5] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. 2008. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *Companion of the 30th International Conference on Software Engineering (ICSE Companion '08)*. ACM, New York, NY, USA, 925–926. https://doi.org/10.1145/1370175.1370190

[6] Paul Hudak. 1998. Domain-specific languages. In *Handbook of Programming Languages*. Vol. 3. Macmillan Technical Publishers, Indianapolis, 39–60.

[7] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. 2017. Reliable and Automatic Composition of Language Extensions to C: The ableC Extensible Language Framework. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 98 (Oct. 2017), 29 pages. https://doi.org/10.1145/3138224

[8] Lennart C.L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 444–463. https://doi.org/10.1145/1869459.1869497

[9] Donald E. Knuth. 1968. Semantics of context-free languages. *Mathematical systems theory* 2, 2 (01 Jun 1968), 127–145. https://doi.org/10.1007/BF01692511

[10] Filip Strömbäck. 2017. *A Syntax Highlighting and Code Formatting Tool for Extensible Languages*. Master's thesis. Linköping University, Linköping, Sweden. http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-138847

[11] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages As Libraries. *SIGPLAN Not.* 46, 6 (June 2011), 132–141. https://doi.org/10.1145/1993316.1993514

[12] Eric Van Wyk, Lijesh Krishnan, Derek Bodin, and August Schwerdfeger. 2007. Attribute Grammar-Based Language Extensions for Java. In *ECOOP 2007 – Object-Oriented Programming*, Erik Ernst (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 575–599.