The Ideal Parser Generator

Ivan Kochurkin
Positive Technologies
Moscow, Russia
ivan.kochurkin@gmail.com

CCS Concepts • **Software and its engineering** → *Compilers*; *Source code generation*;

ACM Reference Format:

Ivan Kochurkin. 2018. **The Ideal Parser Generator. In *Proceedings of SLE 2018*. ACM, New York, NY, USA, 2 pages.**

1 Token value comparison operator

Many languages, such as SQL dialects, have a large number of words that are keywords only in particular contexts. The standard solution is to put all such keywords in an identifier rule:

```
identifier : ID | GET | /* other keywords */
```

Although this solution is broadly applicable, it has drawbacks: such keywords in a particular context could be numerous, and it is easy to forget one by mistake. Moreover, if the number of keywords is large, parser performance can suffer.

Another way is to get the value of an identifier token and recognize the relevant rule based on it. This requires a code insert, which unfortunately limits the scope of grammar application to the Java runtime.

```
getter
    : {_input.LT(1).getText().equals("get")}?
    Identifier propertyName;
```

To avoid Java dependence and make the grammar truly universal, we could introduce a special construction for comparing a token's value with that of another token:

```
getter : Identifier=='get' propertyName;
```

This feature has been previously proposed on GitHub: [1].

2 Universal code inserts

Certain syntactic constructions cannot be covered even by additional syntax and parsing them correctly requires computations. Examples include in C# or PHP [2]. As mentioned already, *semantic predicates* or *actions* can be used for such computations in ANTLR. However, they are not universal.

With certain tricks, we can generalize them to multiple runtimes, as has been done in the JavaScript ANTLR grammar [3]. However, this approach is not ideal since abstracting the parser is an additional stage, and a complicated one at that. Such code inserts are foreign and integrate poorly with generated code, since the code is not checked during generation of the parser from the grammar and is poorly formatted. In addition, grammar IDEs do not work well with inserts.

A **Universal DSL** for describing semantic predicates would solve all these problems, in a laconic and elegant way. By combining a declarative approach (context-free grammar) and imperative approach (universal inserts), we could simplify creation of all sorts of parsers!

For instance, the instruction `la(-1)` for getting the previous character is translated as `_input.LA(-1)` in the Java runtime and as `self._input.LA(1)` in the Python runtime. Read more about universal code inserts on GitHub [4].

3 Full-fidelity parse tree

ANTLR has a channeling mechanism for isolating a set of hidden tokens (spaces, comments) from the main tokens. However, these tokens are associated with nodes on the parse tree. Besides hidden symbols, there are unrecognized (error) symbols for which no lexical rules matched, and error tokens for which no grammar alternatives matched.

It is very important that such tokens be **associated** with nodes of the parsing tree: this structure makes it possible to take code style into account during transformation and refactoring, for example. Plus by analyzing just the parse tree and spaces, we can detect serious vulnerabilities in source code, such as goto fail1 [5].

More on GitHub: [6], [7].

4 Idiomatic generated code

In ANTLR, the first letter of *lexical* rules is capitalized, whereas the first letter of *parser* rules is in lower case. Target runtimes each have their own code style, which is completely ignored during rule transformation. Worse still, the generated code may be actually invalid because certain identifiers may be reserved for keywords. This fact creates a choice: either sacrifice universality, or try to fit the target runtime by renaming rules.

SLE 2018, November 2018,

Instead of changing rules in the grammar itself, we could adapt identifiers for each target runtime. Therefore the grammar is completely independent of the runtime and Visitor code will look runtime-appropriate.

More on GitHub: [8], [9].

5 Scannerless parsers

For certain languages and syntactic constructions, the stages of tokenization, parsing, and semantic analysis are heavily intertwined.

One example: in C languages, the construction `x * y` is ambiguous. It can be interpreted as multiplication or as declaration of a pointer. This ambiguity is resolved with the help of a symbol table.

In the popular Markdown text format, parsing errors do not exist. Rather, every symbol is interpreted either as a formatting element or as regular text. If we write: `[link](https://google.com)`, we get a link. If we remove the closing parenthesis, then we get plain text: `[link](https://google.com/`.

Description on GitHub:[10].

6 Online grammars editor and browser

Plugins for popular IDEs can be used for developing and debugging of ANTLR grammars. Plugins include ANTLR intellij-plugin-v4 [11] and [12]. However, they do not enable developing grammars without an IDE or for all runtimes simultaneously, and most importantly, they work offline. So in order to test even something small, the user must install and configure additional software.

It would be much more convenient to test grammars online right in a web browser. Users could choose to create their own language or use an existing grammar from the ANTLR official repository [13] to view the parse tree for a particular code fragment. The ability to make fixes then and there, and add them to the repository, would make things even more convenient.

On AST Explorer [14], users can build a parse tree only for certain languages (JavaScript and CSS), and cannot change the grammar.

To demonstrate this concept, I have developed an open-source desktop application: Desktop Antlr Grammar Editor (DAGE) [15]. The left side of the window allows editing the grammar, while the right side allows editing the language. Both the grammar and code can have associated errors at the relevant levels of parser generation.

7 Conclusion

The proposed features expand upon the abilities of standard context-free grammars, as well as unify the API and generated code. This should make development of ANTLR-based parsers more convenient.

A web grammar editor would automate and streamline the development process, thus reducing the domain barrier to entry.

These changes would also attract new developers working with various levels and languages. Beginners could quickly and easily get to work on making their own grammars. Those with more ANTLR experience could help to quickly fix issues in existing grammars. And what's more, grammar would become universal and no longer be runtime-dependent.

References

[1] Token value comparison operator, https://github.com/antlr/antlr4/issues/1965.

[2] C# string interpolation, https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/tokens/interpolated.

[3] Universal JavaScript Grammar, https://github.com/antlr/grammars-v4/tree/master/javascript.

[4] Universal actions language, https://github.com/antlr/antlr4/issues/1045.

[5] Anatomy of a "goto fail", https://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch/.

[6] Create parse trees whose leaves have left/right hidden token text, https://github.com/antlr/antlr4/pull/1667.

[7] Error recovery should create missing tokens for first path to recovery point, https://github.com/antlr/antlr4/issues/1972.

[8] Add a new option for naming convention of generated Visitor and Listener methods, https://github.com/antlr/antlr4/issues/1615.

[9] Throw warning if symbol conflicts with generated code in another language or runtime, https://github.com/antlr/antlr4/issues/1670.

[10] Add scannerless tokenizer to runtime jar, https://github.com/antlr/antlr4/issues/814.

[11] ANTLR intellij-plugin-v4,https://plugins.jetbrains.com/plugin/7358-antlr-v4-grammar-plugin.

[12] vscode-antlr4, https://marketplace.visualstudio.com/items?itemName=mike-lischke.vscode-antlr4.

[13] ANTLR grammars repository, https://github.com/antlr/grammars-v4.

[14] AST Explorer, http://astexplorer.net/.

[15] DAGE, https://github.com/KvanTTT/DAGE.