Event-Based Parsing

Vadim Zaytsev
Raincode Labs
Brussels, Belgium
vadim@grammarware.net

Abstract

Event-based parsing is a largely unexplored problem. Despite several hugely popular event-based parsers like SAX, there is very little research on the ways grammar engineers can be given explicit control over handling input tokens, and the consequences of exposing this control. Tool support is also underwhelming, with no language workbenches and very few libraries to help a parser developer to get started quickly and efficiently. To explore this paradigm, we have designed a language for event-based parsing and developed a prototype that translates specifications written in that language, to parsers in C#. We also report on the comparative performance of one of the parsers we generated, and a previously used PEG parser extracted from a real compiler.

CCS Concepts • **Theory of computation → Parsing**; • **Applied computing → Event-driven architectures**.

ACM Reference Format:
Vadim Zaytsev. 2019. Event-Based Parsing. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS '19), October 21, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3358503.3361275

1 Introduction

Parsing is considered a solved problem [1]. However, in practice often it is not. Despite having literally hundreds of different parsing techniques at our disposal [9], produced by the researchers and practitioners non-stop since 1961 [10], the compiler experts are commonly faced with challenges related to inapplicability of existing technologies to the tasks of software renovation [2], the inappropriateness of existing frameworks in dealing with legacy languages [29] or simply the lack of developed theories and tools for crucial activities like regression parsing [28].

In general, parsing in a broad sense [32] is a task of recognising elements of expected structure in the input stream. There are many flavours of such techniques, forming a spectrum from classical text-to-tree parsing techniques [9] to a family of more approximate and tolerant semiparsing techniques [27] all the way to the simplest tasks of software analytics [3] and software metrics [5, 19]. On the grand scheme of things, counting the number of lines in a file is also some form of "parsing" (more commonly referred to as "fact extraction"). As an industrial company involved in writing compilers and migrating legacy software, we routinely encounter new challenges in parsing. For example, some notations of legacy languages are position-based [29], and "parsing" entails counting which position in the line does a character occur at, and not necessarily paying any attention to the character per se (and counting the number of spaces in a line before a non-space symbol has much more in common with counting lines in a file than with traditional graph manipulation).

This paper is an attempt to explore a new paradigm in parsing: the event-based parsing. Instead of writing a grammar for the desired language, typically specifying rules like "`a 'b' c+`", meaning "sequentially apply the rules of the nonterminal `a`, then expect an input `'b'`, and then expect any number of inputs conforming to the rules of the nonterminal `c`", we could write a *reactive* specification in the form of "whenever `'b'` is found in the input, expect `a` to have been prepared before it, and collect any number of occurrences of `c` until the input is exhausted".

To quote Tudor Gîrba: "In software ideas do not exist without a concrete incarnation. The materialization of an idea is a step that matters and the research is not complete without it." [8]. Contemplating novel paradigms is always easier with a concrete implementation of them, even though, of course, we are thus inherently limiting ourselves to the limitations of the actual implementation at hand. Thus, we will present *Engage!* [31] as a small framework supporting writing parsing specifications in an event-based style, and generating code in C# for execution and inspection.

Motivations for choosing the event-based paradigm can be versatile. At least two possible advantages come to mind in the context of parsing. First of all, event-based representations are equally easy to write when precise parsing is required, as well as when some form of semiparsing (tolerant, error-correcting, permissive, fuzzy, etc [27]) is enough. The state of the art in traditional state-based parsers is that most effort goes into tool support for precise parsing, and each language workbench which can already deliver precise parsers, introduces some idiosyncratic ways of injecting tolerance directly into the grammars (or other forms of parsing specifications). For example, in the language of the case study we will use below, a string type is represented as `char(n)` where `n` is a literal number. The simplest event-based specification for collecting all string types used within a program, in *Engage!* would look like this:

```
'dcl'    -> push DclBlock(types)
            where types := await* String
'char'   -> push String(n)
            where n := await number
'enddcl' -> trim String
```

(The language will be explained in the next section, but for now the intuitive understanding of this little example is enough). This specification is tolerant—it does not check for matching parenthesis, ignores other types, skips over statements,—so what it produces is a semiparser. However, it can be extended to make a full parser by lifting and dropping flags to signal changes of state, and adding appropriate actions to react to opening and closing brackets.

The second possible reason to actively prefer event-based parsers as opposed to state-based is the ease to produce an online algorithm (i.e., one that acts on token-per-token or symbol-per-symbol basis and does not require the entire input to start parsing). Some traditional parsing techniques are easier to convert to an online variant—those that follow the input (the common term is "bottom-up")—than others that follow the grammar ("top-down") while parsing, even though top-down online algorithms exist, such as OMeta [24]. However, the machinery often needs a buffer to fake the onlineness: many parsers rely on a feature called "lookahead" and make a decision based on the next token/symbol and several tokens/symbols following it. In an event-based specification scheduling some expectations comes naturally and do not break the basic paradigm of decision making based on one current symbol or token. We acknowledge this need but do not pursue it in detail further in this document.

In the rest of the paper, we present a prototype system for generating parsers from event-based specifications, called *Engage!*: section 2 explains the input language of *Engage!* and the reactive commands it accepts, and unveils some details about their implementation, which might seem insignificant but turned out to have a measurable impact on the effectiveness of the system; section 3 describes the case study and the framework for evaluating the resulting parsers, by comparing a parser generated by *Engage!* to a previously existing parser which is a part of one of our industrial compilers; section 4 presents the data obtained from the experiments, and draws preliminary conclusions from the lessons learnt there; section 5 refers to a few literature points with ideas similar to those proposed here; section 6 concludes the paper.

2 Language

The infrastructure of the parser generated by *Engage!* is rather simple. It includes an online tokeniser that slides further into the input stream and creates the new typed token for the parser to consume, the parser, the main stack of already manufactured objects and a queue of pending actions. On each iteration, the parser takes the next token, decides which actions to undertake based on the type and the value of the token, which include pushing new values onto the stack, popping elements from the stack, observing the type of the top of the stack, scheduling new actions, triggering them, etc.

Each rule has a form of $t \longrightarrow a$, it contains a left hand side which is a trigger t and a right hand side with an action a. The trigger is usually a token or a token type, and possibly a flag. There are several predefined token types, but the user can define any number of them if needed:

- ◇ `skip` is a token type that does not trigger an event, these are characters skipped by the tokeniser which also uses them to find lexeme boundaries. A typical declaration would be "`' ', '\t', '\r', '\n' :: skip`", telling the tokeniser to skip spaces, tabs and newlines.
- ◇ `mark` is a token type given to punctuation marks like brackets or commas, which always produce their own tokens even if not separated by a skippable token.
- ◇ `word` is a token type for reserved words, this is usually the most detailed and populated type in a specification, all keywords normally go here. Example: "`'if', 'map' :: word`".

As for the tokens themselves, they are usually represented literally as single-quoted strings. There are also two inferred token constructors: `number` which matches any non-empty sequence of digits, and `string` which matches a non-empty sequence of non-skippable characters. Any token that was defined to belong to the token type `word`, is not matched by `string`.

The following list includes the possible actions as they are written in the specification (the parser generator goes through several phases of internal representations before generating actual code of the parser in C#):

- ◇ `push` is the main action for creating new objects and putting them on the main stack for further consumption. It is often accompanied with a `where` clause to separate actual object creation from collecting its constituents. For example, "`push A(x) where x := await B`" will instruct the framework to wait until an object of type `B` is put on the stack, and then pass it as a parameter to a constructor of type `A`, replacing the `B` object at the top of the stack with the newly created `A` object. Neither the arguments of the constructor nor the `where` clause, are mandatory, if the object can be created without arguments.

- `wrap` is an action equivalent to a combination of `push` with one argument and a corresponding `pop`, with one important exception of not being mandatory. If a `push` action cannot complete its object creation, it fails the parsing process with an error; if a `wrap` action cannot complete its object creation, it silently terminates and evaporates from the list of pending actions. The `wrap` action can be used to clean up the stack when an end of a section is encountered.
- `tear` is the opposite of `push`: this action takes a previously created object of an expected type and decomposes its constituents into named entities that can be used within the parent action. We found it useful to have to "undo" object creation which was made erroneously. Obviously, unhealthily enthusiastic reliance on `tear` can damage the performance of the parser, but in some scenarios it is easier to create the object first and then undo it in one or two special cases, instead of listing near-exhaustively each reason for its creation. In our evaluation grammar we used it to get back to trivial types like `int` or `string` from boxing types representing numbers and identifiers.
- `drop` is a trivial action of taking the top of the stack and disregarding it. It can be viewed as a more aggressive form of undoing.
- `pop` is a useful action of taking the top of the stack to be used elsewhere. It is not found on the top level (right after the trigger, like `push`), but as a part of the `where` clause. For example, "`';' -> push Decl(v,t) where t := pop Type, v := pop Var`" is the rule we use to create a variable declaration after witnessing its terminator symbol. If all constructs of the language in question, use explicit terminator symbols, the event-driven parser will be the easiest to write and debug, since it will basically fall back on exclusively pushing and popping objects in the right order. The `pop` action respects subtyping: i.e., if the type of the object pushed on the stack, is a subtype of the expected object, the action completes successfully. If the type of the top of the stack is of unrelated type, a parse error occurs.
- `pop*` action represents a sequence of `pop` actions, followed by reversing the order of the result and returning a proper list of objects. For example, in a Pascal-like language we could have written a rule "`':' -> push Decl(vs,t) where vs := pop* Var, t := next Type`" to use the separator symbol to collect the list of the already processed variable names and create a declaration object once the type is processed as well.
- `pop#` action was not anticipated in the initial design of the *Engage!* language but was deemed to be a valuable addition later in the project. Basically if works the same way as `pop*`, and when present alone, generates exactly the same parser code. However, if several `pop#` actions are used within the same `where` clause, they collect their objects together in parallel instead of doing it sequentially. The `pop#` actions are useful in cases where a definition of a class, for example, in Java-like languages, consists of a mixture of field declarations and method declarations, but in the desired representation we like to have them collected in two separate lists. In the case study language we had the same situation with local declarations and executable statements: declarations are allowed everywhere but have global scope, so their positioning only serves readability purposes. Thus, the compiler is allowed to bundle all declarations together while bundling all the statements separately.
- `next` is the first action we describe that implies scheduling an action. It behaves the same way as the `pop` action, but delays its execution to the next object that will be pushed on the stack after the pending action is scheduled. If the `next` action cannot be completed (for example, because of the type incompatibility), then it fails the parsing process.
- `next*` acts like a sequence of `next` actions terminated by the first object pushed on the stack that does not match the expected type (if the first object after scheduling is incompatible, it is possible for `next*` to immediately terminate with an empty list). The main differences with the `await*` action below are the immediacy of the first reaction, and the consecutiveness of the elements collected by the `next*` action.
- `await` is the classical scheduled nested action that binds a name to a value that has not been observed yet. Executing such an action means scheduling a special lambda function handler that will be called at some point in the future once an object of the expected type (or one of its subtypes) is being pushed on the stack. Multiple `await` actions can be combined, in which case the scheduling of the next one is a part of the code that gets scheduled first. A very useful feature to provide the right context to the scheduled actions, is a flag that gets lifted before scheduling and dropped after the scheduling action is successfully triggered. The syntax is: "`x := await T with F`" where `T` is the type to trigger the scheduled action and `F` is a flag. The user can specify additional flags that are to be expected by the waiting action, such as: "`await (Lit upon BRACKET) with CHAR`". This would specify to raise the `CHAR` flag and schedule an action that will only react to pushed objects of type `Lit` if the flag `BRACKET` is set, and will take down the `CHAR` flag once finally fired. In general, flags model state.
- `await*` is an asynchronous counterpart of `next*` which schedules some code that keeps reacting to any object of a compatible type that is attempted to be pushed on the stack. There are three very important points in implementing this action:

   – Quite often such pending actions that tend to "reschedule" themselves, so to speak, are implemented verbatim, with scheduling code being included as the part of the handler. However, according to our experiments, not only did it prove to be an awkward way of writing code in C# (since the lambda function can be defined and scheduled simultaneously but a self-rescheduling function must be declared first as a `null`, and then redefined with an actual definition—it is a well-known pattern in C# [14] but that does not legitimise it as graceful), but also it is very bad performance-wise. Instead, we changed the framework such that the handler can have a return code to express its intention, and the machinery around it takes this return code and either removes the handler from the pending actions list or does not. As a side effect, this allows to combine the wishes of the handler with the wishes of the framework. For example, when the parsing is finished and all remaining handlers are triggered forcefully, none of them are rescheduled even if they wish to continue parsing.
   – By the very nature of this action, which reacts to objects of a particular type and ignores other objects being pushed, it can operate forever. This is a drastic conceptual difference between `await*` and `next*`, due to the latter expecting consecutive objects and thus having a natural terminating condition. Thus, a special `trim` action (discussed below) is often used in combination with `await*`.
   – The "dangling else" problem [11] is solved here automatically by the right order of scheduling and trimming actions. In languages using terminator symbols instead of statement block delimiters (most 4GLs get that from COBOL), it is common to have a pair of terminals like `'if'` and `'endif'` between which all the statements are collected into the branch. Thus, if we have several levels of nested conditional statements, it could be that we will have several coexisting pending handlers for the same type ("statement"), of which only the latest must be fired until it is somehow terminated: either explicitly with a `trim` action, or forcefully at the end of the input together with all other pending actions.
- ◇ `trim` is the action to stop any other ongoing actions. Its most common use is to terminate `await*`, but it can also be used to cancel a simple `await` action or even some variations of the `next`/`next*` actions. For example, in a language with C-like declarations (i.e., "`byte x,y,z;`") we can react on the type and collect the variables by a `next*` action but terminating it at a semicolon.
- ◇ `lift` is the action to activate a flag. The flags were mentioned before when introducing `await`, and here is an action to deal with them directly. The *Engage!* framework does not differentiate between auto-lifted flags by a `with` clause of `await` or `await*` actions, and "manually" handled flags with `lift` and `drop`, so the users can decide for themselves whether to reuse flags.
- ◇ `drop` is the opposite of `lift`, it is an action to take down a given flag. Implementation-wise each flag is just a Boolean variable inside the parser that indicates a particular context and gets assigned `true` or `false` depending on the course of the parsing process. All the `upon`-based guards help to structure the parsing process by using those flags.

Engage! has one additional feature that is unrelated to the event-based paradigm but still useful: it generates all the data structures that its generated parser populates. There is enough information in the constructors used by the input specification to infer all the used data types, and the rest can be added manually since the generated data classes are deliberately made `partial`.

3 Evaluation

Since event-based parsing is largely an unexplored area, we need to set up some experiments to evaluate how it behaves under different circumstances. In order to do that, we have implemented an event-based specification for a fourth-generation language used on mainframes, named AppBuilder. Before that we have recently implemented a full-fledged compiler for that language, named TIALAA, for "There is a Life After AppBuilder" [17], facing quite a number of challenges in the process [28], including writing a dedicated parser generator for one of the notations [29] and building a substantial testing framework [30]. For this paper, we chose a different notation—the AppBuilder ecosystem consists of many different parts, but there are five main notations used for rules, sets, panels, HPS bind files with views and HIS bind files with stored procedures, and for the rest of the paper we focus on "the rule language" of AppBuilder. In order to align the sometimes peculiar and uncomfortable legacy software terminology [17, 30] with presumably modern-minded readers, we will from now on call AppBuilder rules "programs".

This way, we can operate on a real life example: the compiler we wrote, is currently in the last phases of the quality assurance process at the customer side, and is about to go in production soon. We will also be able to make direct comparisons with the existing and well-tested parser. Our "old" parser was written in the style of Parsing Expression Grammars (PEG) [7], and was produced by our own in-house developed parser generator, which closely follows the *Packrat!* algorithm [6] with left recursion [23], memoisation and backtracking, and some extra features not found in alternative generators but unrelated to the event-based story. The original documentation of AppBuilder exists, but could not be consulted by us for legal reasons, the same goes for the baseline implementation of it. Thus, we deliberately limit ourselves to the language of TIALAA as a reasonable proxy (the only claim we can make with certainty is that it is a superset of the language used by our customer in their codebase containing almost 4 MLOC).

The syntax of the chosen language is a perfect example to try out this new parsing framework, because it was designed without following any principles of systematic software language design. For example, it contains declaration blocks and statement blocks, yet declaration blocks have start and end delimiters and statement blocks do not; each declaration is terminated with a semicolon, yet statements are not separated by anything; conditions and `case` branches do not separate the condition from the body, yet `case` branches allow for multiple values to be present per branch, leading to constructs like `case a b c` where `a` and `b` might be values or calls to procedures that return such values, and `c` may be a procedure call inside the branch; two consecutive string literals are concatenated as if linked by a + operator, unless they occur in the begin of the `case` statement, where two adjacent string literals are treated as two separate matching expressions for the branching statement; there are two printing statements with almost equivalent semantics differing on two seemingly innocent built-in data types out of 15; there is signature-based polymorphism in procedures, yet the rest of the language does not have name uniqueness, and there can be peaceful coexistence of a rule, a view, a set, a set value and a panel with exactly the same name; complex data structures can be defined inline inside a program in a declaration block or encoded in a separate HPS bind file, yet each style allows to define structures that cannot be expressed in the other; most statements do not expect brackets around their arguments, yet for one of them they are mandatory. The list can be extended, but even this humble collection of AppBuilder trivia should be convincingly showing that implementing such language is way beyond repetitive application of the same simple pattern, and thus stresses the design of *Engage!* itself, which was a part of our motivation.

We decided to not align the data structures generated by both parsers, which would have been an extremely labour-intensive process. Thus, we cannot guarantee that the actual parse trees coming from them, are equivalent (we know for a fact that they are not identical). However, we can perform certain sanity checks like the number of statements in a parse tree and their approximate types, and have performed them when testing *Engage!*

The original PEG [7] parser was never intended to be the epitome of performance, but it was reasonably optimised whenever possible. Concretely, in October 2017 we have reported that the recompilation of the entire codebase of our customer was taking 48 hours [28, §2.1], while now in September 2019 it only takes 6 hours (8× speedup).

To test and stress test our parsers and see how they perform against each other, we have designed and generated the following types of test programs (in each of them N is a size parameter which usually went from 0 to 1000, these results will be reported on and visualised on the next pages; and additionally for true stress testing we produced test programs with $N = 10^k$, for k from 0 to 8):

- **Long** tests are the most straightforward and gentle form of stress testing: each simply contains a flat sequence of N statements. For compound statements such as loops or conditional statements, we generate the simplest possible statement (`return`) as contents of each expected block. Such test programs are barely harder to parse than to generate. We do not expect to find any problems due to the size of these tests, because compilers usually scale easily this way (i.e., once individual statements and sequences are implemented and tested, sequences of arbitrary statements work fine in any quantities), but we wanted to have a first baseline comparison of some test data that both parsers should have had no trouble with.
- **Deep** tests are designed to stress test parsers' capability to handle deeply nested constructs in code. Each such program consists of a single conditional statement with a trivial condition and the body consisting of a single similar conditional statement, for a total of N nesting levels.
- **Stacked** tests were extremely useful to both fish out the bugs in the parser generator, and in stress-testing the result. They all have this format:

```
N single random statements
if C_1
    N single random statements
    if C_2
        N single random statements
    endif
    N single random statements
endif
N single random statements
```

  Since the conditional statement was implemented with an `await*` and `trim` combo, it was crucial to test that in case of nesting the right actions get scheduled and trimmed at the right moments, producing a correct tree.
- **Declaration** tests are structured analogously to Long tests, but they start with a declaration block of N random declarations, which is then followed by N random single statements. Just as with Long tests, we expected no challenges here, but we were about to get surprised (see section 4).
- **Mixed** tests have two parameters, N and K, and consist of a mixture of declaration blocks and statement blocks, of the following form:

$$K \text{ times} \begin{cases} \texttt{dcl} \\ \qquad N \text{ random declarations} \\ \texttt{enddcl} \\ N \text{ single random statements} \end{cases}$$

We varied both N and K from 0 to 100.

4 Results

In an attempt to minimise the impact of the C# compiler, .NET Core runtime and other factors, each measured point was obtained as follows. After parsing ten random files and discarding the results as warmup, we read the file once, parse its contents 100 times, recording the execution time (with .NET Core's `Stopwatch`) for each run. Then, 10 fastest and 10 slowest results are discarded, and an average is computed of the remaining 80 measurements. This average forms one data point to be recorded and plotted.

- **Long** tests (Figure 1) exhibit a very unpredictable behaviour of the PEG parser—switching to a different statistic (mean, median, normalised mean, etc) did not help us to smoothen the results. The only hypothesis to explain these measurements is the garbage collector of .NET Core which interfered more with the PEG parser because it was more complex and thus created more disposable objects. Besides the wild spikes, both parsers behave linearly which was to be expected—PEG parsers are known to have near-linear performance in many cases [13, 18]. Yet, the event-based parser was consistently faster, if by a small margin.
- **Deep** tests (Figure 2) were definitely non-linear, but drastically worse for the PEG parser, which exploded with a stack overflow exception at depth 132. Even at the depth 1000, the *Engage!* parser was as fast as the PEG parser at around depth 60. This is attributed entirely to the implementation strategy: the *Engage!* parser being at depth 1000 only means it keeps a flat list of 1000 pending handlers that are ready to be triggered at the next completed statement: neither the handlers nor the parser is toxically recursive on any of this; being at depth 130 for the PEG parser means being at extreme depth of its own recursion, as well as operating normally with other things like memoisation of all the symbols consumed so far.
- **Stacked** tests (Figure 3) were harder for both parsers than the Long tests, but that only raised the execution time lines for both of them slightly upwards. The jitter of the lines (still attributed to garbage collection and other internal processes of the runtime of .NET Core) is slightly milder for the PEG parser and slightly wilder for the *Engage!* parser.
- **Declaration** tests (Figure 4) show the same familiar linear behaviour for the *Engage!* parser, with slightly less jittery data from the PEG parser, but this time clearly tending towards its upper end—which is around 12 times slower.
- **Mixed** tests (Figure 5) produce a line for each fixed N and for each fixed K, but to visualise all the results without the sawtooth effect we sort the measurements by the execution time of the *Engage!* parser. The results are not much faster mostly because the *Engage!* specification is `pop#`-based, not `await*`-based. We are contemplating rewriting it, which would require introducing a new kind of event for the initialisation.

5 Related Work

SAX, or Simple API for XML [15], could arguably be the first practical application of event-based parsing. Its development started in 1997 as an alternative to various state-based parsers for XML that existed at that time. The initiative involved many people, mostly participants of the XML-DEV mailing list, and the first release happened in 1998, only several months after its conception [22]. The latest release of SAX at the moment is 2.0.2, released in 2004. Various conceptual descendants of SAX are in reasonably wide use, such as HTMLParser [16] commonly used to "parse" HTML by converting it to a list of events for starting a tag, ending a tag and having raw data between tags. It is almost functionally equivalent to SAX, with a notable exception of not handing processing instructions. Both SAX and HTMLParser are hand-written event-based parsing libraries, they do not allow their users to create new parsers in the same style the way *Engage!* does, but their success shows that there is interest in online parsing methods. They serve as tangible anecdotal evidence of the relevance of event-based parsing as a technology.

RxParse [4] is a framework for event-based SAX-style parsing for Java. There seem to be no academic publications about it, and the documentation consists exclusively of code examples with errors in them (such as "subscirbe"), but it looks like a library or an internal DSL. While we as a company are in general much less interested in JVM-based solutions as opposed to .NET-based solutions, it would be nice to make a detailed feature-wise comparison of RxParse with *Engage!*There seems to be some interest and some activity in making a similar event-based library for event-based LINQ-style parsing for .NET [21], but at the moment there is no library with responsible maintainers behind it.

The data structures produced by any *Engage!* parser are closer to what would have been defined by a dependency grammar, since each of the statements may have its own behaviour and its own way of arranging its constituents. Recently Steimann was advocating that perhaps that is indeed a better way to define software languages [20]. On the concrete level, he proposed the use of "langrams" which define syntactic categories and contain a lexicon of entries. Each lexicon entry is a triple ⟨lexeme, dependents, semantic

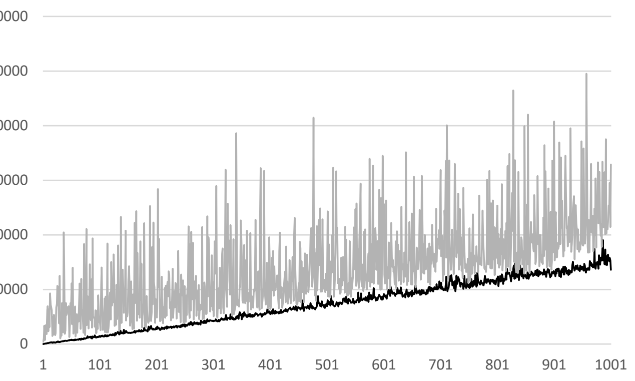

Figure 1. Performance of the *Engage!* parser (lower line) and the PEG parser for the same language on **Long** tests. The input data is generated test programs with a sequence of N oneliner statements, for N from 0 to 1000, horizontally. The vertical axis is in CPU ticks. This experiment took around 4 minutes on the developer's laptop. We have no solid explanation for the erratic behaviour of the PEG parser here, but traditionally blame the garbage collection.

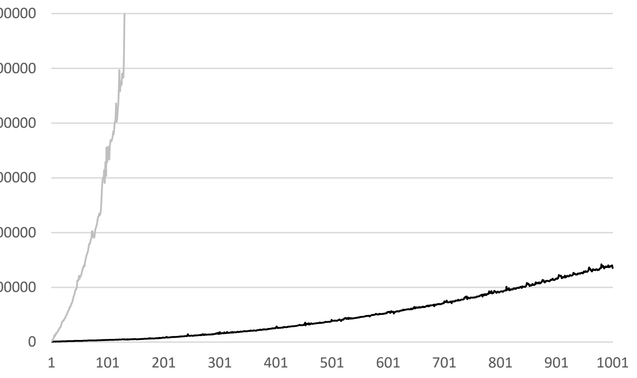

Figure 2. Performance of the *Engage!* parser (lower black line) and the PEG parser for the same language on **Deep** tests. The input data is generated test programs with a single conditional statement containing a single conditional statement, and so forth, for a total of N levels, for N from 0 to 1000, horizontally (the line for the PEG parser stops at 131, larger programs were causing stack overflow). The vertical axis is in CPU ticks. This experiment took around 13 minutes on the developer's laptop.

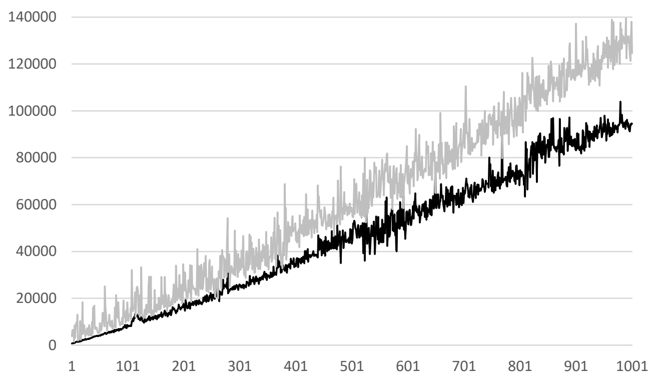

Figure 3. Performance of the *Engage!* parser (lower line) and the PEG parser for the same language on **Stacked** tests. The input data is generated test programs with N random single statements, then a conditional statement containing N random single statements, a nested conditional statement with N random single statements of its own, and N more random single statements, the last conditional statement followed by the last N random single statements, for N from 0 to 1000, horizontally. The vertical axis is in CPU ticks. This experiment took around 19 minutes on the developer's laptop.

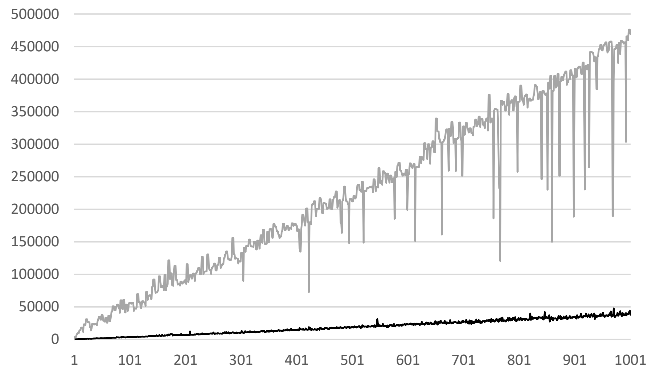

Figure 4. Performance of the *Engage!* parser (lower black line) and the PEG parser for the same language on **Declaration** tests. The input data is generated test programs with N random declarations followed by N random statements, for N from 0 to 1000, horizontally. The vertical axis is in CPU ticks. This experiment took around 34 minutes on the developer's laptop.

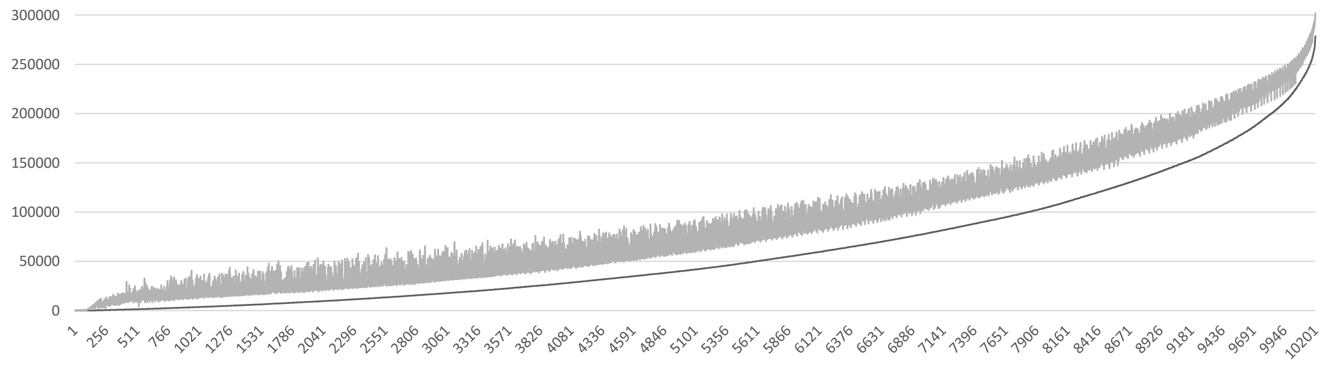

Figure 5. Performance of the *Engage!* parser (lower line) and the PEG parser for the same language. The input data is generated test programs with K pairs of a declaration block with N random declarations and a statement block with N random statements, for N from 0 to 100, for K from 0 to 100, sorted horizontally by the execution time of the *Engage!* parser to avoid sawtooth shapes. The vertical axis is in CPU ticks. This experiment took around 50 minutes on the developer's laptop.

```
namespace AB
types
    ABProgram;
    Integer, String, Decimal <: Type;
    Decl;
    Var, Lit <: Expr;
tokens
    ' ', '\r', '\n' :: skip
    ';', '(', ')' :: mark
    'dcl', 'enddcl', 'integer', 'dec' :: word
    number :: Num
    string :: Id
handlers
    EOF                    -> push ABProgram(data,code)
                              where code := pop# Stmt,
                                    data := pop# Decl
    Num                    -> push Lit(this)
    'dcl'                  -> lift DCL
    'enddcl'               -> drop DCL
    ';' upon DCL           -> push Decl(v,t)
                              where t := pop Type,
                                    v := pop Var
    'integer' upon DCL -> push Integer
    'dec'      upon DCL -> push Decimal(n)
                              where x := await (Lit upon BRACKET) with DEC,
                                    n := tear x
    '(' upon DEC           -> lift BRACKET
    ')'                    -> drop BRACKET
```

Figure 6. An excerpt from `appbuilder.eng` [31], an *Engage!* specification for the rule language of AppBuilder (the language as it is used in the Raincode TIALAA compiler). One can see four main top-level directives, defining the namespace for the generated data classes as well as the parser class; the data types to be generated and populated, including their subtyping relations which would otherwise not be visible from the constructors used in the handlers; then the tokens filling three predefined types and defining two user-defined types with predefined matchers; and finally some of the handlers parsing some of the local field declarations of AppBuilder.

definition⟩. Since his lexemes are typed, they are technically not lexemes but tokens (his prototype transforms them to Prolog where they become untyped again), corresponding to our triggers, and our actions are executable strategies of obtaining dependents and populating the resulting structure.

The reactive paradigm has been explored before in the context of compiler development, as a "pending evolution" of grammars [26]. Within that framework, one could schedule handlers for changes in a grammar, including persistent handlers that would reset after each triggering, performing actions like "if anyone introduces a new nonterminal, transform its name to uppercase".

6 Concluding Remarks

In this paper, we set out on a journey to explore the paradigm of event-based parsing. Traditional parser generators that allow the end user to specify a grammar stating which nonterminal and terminal combinations constitute which other nonterminals, and inferring the parsing actions from it behind the scenes. Contrary to them, we wanted to let the end user write a reactive specification, explicitly stating which actions need to be performed when witnessing a particular token in the input stream. An example of such specification is given on Figure 6. To explore this paradigm, we have developed a parser generator of our own, which takes $\{t \longrightarrow a\}$, a collection of trigger-action pairs, and some metadata, and produces all the necessary data classes, the tokeniser and the parser. The generator is called *Engage!* and partly is made publicly available through GitHub [31]. With anyone willing to use *Engage!* to explore this further, we are willing to collaborate and provide all the missing components that could not be made open source (they only concern the parser of the *Engage!* itself, so at some point could even be made self-sufficient by bootstrapping).

Engage! is not perfect and could have been optimised in many ways (one of the most obvious paths could have been to emit IL code directly instead of generating C#, but we found it quite useful to inspect the generated code. Yet, it clearly shows promising results in both expressing constructs of weird non-orthogonal legacy languages, as well as in outperforming a traditional industrial strength parser by a small margin on simple programs and by a non-linearly growing margin for deeply nested programs. Instead of investing in these possibly premature [12] optimisations, we would like to continue the exploration by adding new action types to the ones introduced in section 2, up to the point where we can think of writing a grammar extraction [25] algorithm converting context-free grammars to event-based parsing specifications and assessing the readability and maintainability of the result. Feedback from the workshop participants could be an immense help to push in that direction.

References

[1] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Gros, A. Kamsky, S. McPeak, and D. R. Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of the ACM*, 53(2):66–75, 2010. doi:10.1145/1646353.1646374.

[2] M. v. d. Brand, M. P. A. Sellink, and C. Verhoef. Current Parsing Techniques in Software Renovation Considered Harmful. In *Proceedings of the Sixth International Workshop on Program Comprehension*, pages 108–117. IEEE Computer Society, 1998. doi:10.1109/WPC.1998.693325.

[3] R. P. L. Buse and T. Zimmermann. Analytics for Software Development. In G.-C. Roman and K. J. Sullivan, editors, *Proceedings of the Workshop on Future of Software Engineering Research (FoSER 2010)*, pages 77–80. ACM, 2010. doi:10.1145/1882362.1882379.

[4] A. Chen. RxParse: Reactive Parse. GitHub, https://github.com/yongjhih/RxParse, 2015.

[5] N. E. Fenton and M. Neil. Software Metrics: Roadmap. In A. Finkelstein, editor, *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, pages 357–370. ACM, 2000. doi:10.1145/336512.336588.

[6] B. Ford. Packrat Parsing: Simple, Powerful, Lazy, Linear Time, Functional Pearl. In M. Wand and S. L. P. Jones, editors, *Proceedings of the Seventh International Conference on Functional Programming*, pages 36–47. ACM, 2002. doi:10.1145/581478.581483.

[7] B. Ford. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In N. D. Jones and X. Leroy, editors, *Proceedings of the 31st Symposium on Principles of Programming Languages*, pages 111–122. ACM, 2004. doi:10.1145/964001.964011.

[8] T. Gîrba. I very much subscribe to this! in software ideas do not exist without a concrete incarnation. the materialization of an idea is a step that matters and the research is not complete without it. Twitter, https://twitter.com/girba/status/1162588504154693632, Aug. 2019.

[9] D. Grune and C. J. H. Jacobs. *Parsing Techniques — A Practical Guide*. Addison-Wesley, second edition, 2008.

[10] E. T. Irons. A Syntax Directed Compiler for ALGOL 60. *Communications of the ACM*, 4(1):51–55, Jan. 1961. doi:10.1145/366062.366083.

[11] A. F. Kaupe. A Note on the Dangling else ALGOL 60. *Communications of the ACM*, 6(8):460–462, Aug. 1963. doi:10.1145/366707.367585.

[12] D. E. Knuth. Structured Programming with `go to` Statements. *ACM Computing Surveys (CSUR)*, 6(4):261–301, Dec. 1974. doi:10.1145/356635.356640.

[13] N. Laurent and K. Mens. Parsing Expression Grammars Made Practical. In *Proceedings of the Eighth International Conference on Software Language Engineering*, pages 167–172. ACM, 2015. doi:10.1145/2814251.2814265.

[14] Matt. Can an anonymous method in C# call itself? StackOverflow, https://stackoverflow.com/questions/1208703/can-an-anonymous-method-in-c-sharp-call-itself, 2009.

[15] D. Megginson. Simple API for XML. Megginson Technologies, http://www.megginson.com/downloads/SAX/, 1998.

[16] Python Software Foundation. HTMLParser — Simple HTML and XHTML parser. https://docs.python.org/2/library/htmlparser.html, 2019.

[17] Raincode. The Raincode TIALAA Compiler. https://www.raincode.com/technical-landscape/tialaa/, 2018.

[18] R. R. Redziejowski. Parsing Expression Grammar as a Primitive Recursive-Descent Parser with Backtracking. *Fundamenta Informaticae, Special Issue on Concurrency Specification and Programming*, 79(3–4):513–524, Feb. 2008.

[19] J. Saraiva. A Roadmap for Software Maintainability Measurement. In D. Notkin, B. H. C. Cheng, and K. Pohl, editors, *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 1453–1455. IEEE / ACM, 2013. https://dl.acm.org/citation.cfm?id=2487035.

[20] F. Steimann. Replacing Phrase Structure Grammar with Dependency Grammar in the Design and Implementation of Programming Languages. In E. Torlak, T. van der Storm, and R. Biddle, editors, *Proceedings of the International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 30–43. ACM, 2017. doi:10.1145/3133850.3133859.

[21] S. C. Taylor. Reactive Parser Combinators. Microsoft Developer Network Forums, https://social.msdn.microsoft.com/Forums/en-US/0f72e5c0-1476-4969-92da-633000346d0d/reactive-parser-combinators, 2010.

[22] J. Tigue, P. Murray-Rust, T. Bray, D. Megginson, and D. Brownell. SAX Genesis. Megginson Technologies, http://www.saxproject.org/sax1-history.html, 1998.

[23] A. Warth, J. R. Douglass, and T. D. Millstein. Packrat Parsers Can Support Left Recursion. In *Proceedings of the 13th Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 103–110. ACM, 2008. doi:10.1145/1328408.1328424.

[24] A. Warth and I. Piumarta. OMeta: An Object-oriented Language for Pattern Matching. In P. Costanza and R. Hirschfeld, editors, *Proceedings of the Symposium on Dynamic Languages (DLS)*, pages 11–19. ACM, 2007. doi:10.1145/1297081.1297086.

[25] V. Zaytsev. Notation-Parametric Grammar Recovery. In A. Sloane and S. Andova, editors, *Post-proceedings of the 12th International Workshop on Language Descriptions, Tools, and Applications (LDTA 2012)*. ACM Digital Library, June 2012. doi:10.1145/2427048.2427057.

[26] V. Zaytsev. Pending Evolution of Grammars. In J. De Lara, D. Di Ruscio, and A. Pierantonio, editors, *Post-proceedings of the Second Workshop on Extreme Modeling (XM 2013)*, volume 1089 of *CEUR Workshop Proceedings*, pages 28–35. CEUR-WS.org, Oct. 2013. http://ceur-ws.org/Vol-1089/4.pdf.

[27] V. Zaytsev. Formal Foundations for Semi-parsing. In S. Demeyer, D. Binkley, and F. Ricca, editors, *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE 2014 ERA)*, pages 313–317. IEEE, Feb. 2014. doi:10.1109/CSMR-WCRE.2014.6747184.

[28] V. Zaytsev. Open Challenges in Incremental Coverage of Legacy Software Languages. In L. Church, R. P. Gabriel, R. Hirschfeld, and H. Masuhara, editors, *Post-proceedings of the Third Edition of the Programming Experience Workshop (PX/17.2)*, pages 1–6, 2017. https://dl.acm.org/citation.cfm?id=3167105.

[29] V. Zaytsev. Parser Generation by Example for Legacy Pattern Languages. In M. Flatt and S. Erdweg, editors, *Proceedings of the 16th International Conference on Generative Programming: Concepts and Experience (GPCE)*, pages 212–218. ACM, 2017. doi:10.1145/3136040.3136058.

[30] V. Zaytsev. An Industrial Case Study in Compiler Testing. In D. J. Pearce, T. Mayerhofer, and F. Steimann, editors, *Proceedings of the 11th International Conference on Software Language Engineering (SLE)*, pages 97–102. ACM, 2018. doi:10.1145/3276604.3276619.

[31] V. Zaytsev. Engage! GitHub, https://github.com/grammarware/engage, 2019.

[32] V. Zaytsev and A. H. Bagge. Parsing in a Broad Sense. In J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, editors, *Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014)*, volume 8767 of *LNCS*, pages 50–67. Springer, Oct. 2014. doi:10.1007/978-3-319-11653-2_4.