Token Interception for Syntax Embedding

Tijs van der Storm[1,2]

[1] Centrum Wiskunde & Informatica
[2] Universiteit van Amsterdam
Amsterdam, The Netherlands
storm@cwi.nl

Abstract. Syntax embedding is an attractive implementation pattern for the development of domain-specific languages (DSLs). However, parsing a source text containing different embedded DSL fragments currently requires complicated techniques and tools. In this paper I present token interception as a simple and elegant implementation pattern for implementing syntax embedding. The resulting architecture supports general, multi-level and dynamic embeddings of syntax. The pattern is used in the implementation of context-free syntax embeddings in Clojure.

1 External DSLs Internally

Embedding is an attractive DSL implementation pattern because it allows the facilities and libraries provided by the host language to be reused [14]. Yet, the syntax of such internal DSLs is constrained by the semantic and syntactic flexibility of the host language. External DSLs, on the other hand, are implemented using elaborate tools such as parser generators, attribute grammar systems, or transformation systems. As a result DSLs implemented this way may enjoy complete syntactic freedom, but this comes at the cost of host language integration.

Syntax embedding comfortably sits between these two worlds. The objective is to combine the host language integration of embedding, with the syntactic freedom of provided by external language tools.

There are many techniques for implementing syntax embedding. One approach consists of statically combining the grammars of host language and embedded language and then use that grammar to parse the source code that contains the embedded fragments [5, 6, 12]. A drawback of this approach is that the syntax of an embedded language has to be processed external of the main program itself and there is a separate step to generate the parser for the combined language. Another approach is to use a extensible parsing algorithm to dynamically modify the "current" grammar during parsing [1, 3, 4, 7–9, 13].

In general, these algorithms are either rather complicated or restricted because it is hard to scope the effects of modifying a parser at run time. The class of allowed grammars is often restricted to LL(k) [10], PEG [13, 17], or some specific class [4] in order to better understand the effects of changing the parser during parsing. Grammars thus have to be written in a specific form, for instance in order to remove left recursion. More powerful algorithms such as used in [1, 9], require strategies or heuristics to deal

```
(ns stm)
(use 'metagrammar)
(def *stm-grammar* #grammar(
   Decls    ::= stm:Decl*;
   Decl     ::= state:State ;
   Decl     ::= actions:Actions;
   Decl     ::= commands:Commands;
   Decl     ::= events:Events;
   State    ::= "state" id Actions? Trans* "end";
   Actions  ::= "actions" "{" id+ "}";
   Trans    ::= trans: id "=>" id;
   Commands ::= "commands" Command+ "end";
   Command  ::= command: id id;
   Events   ::= "events" Event+ "end";
   Event    ::= event: id id;))
```

Fig. 1: Concrete grammar for a state machine language

with ambiguous parses. It is often unclear whether it is possible to introduce ambiguities between host language and embedded language.

In this paper I present token interception, an implementation technique for syntax embedding that is simple to understand, and flexible enough to allow a large class of languages to be embedded. It does not suffer from the problems indicated above. The idea is to see tokenizer and parser as co-routines that can be suspended and resumed upon intercepting certain tokens. During suspension different tokenizers and/or parsers can be given control to hijack parts of the input stream. The pattern has been used in the implementation of SYNFUL LISP, an extension of Clojure that allows dynamic, multi-level embedding of languages defined by arbitrary context-free grammars[3].

Organization First, Section 2 presents an overview of the embedding capabilities of SYNFUL LISP. Then, in Section 3 the token interception implementation pattern is presented in generic terms. After that, the implementation of SYNFUL LISP using token interception is described in Section 4. The approach is then evaluated in a lightweight assessment. Finally, the paper is concluded with final remarks.

2 SYNFUL LISP

2.1 Introduction

SYNFUL LISP is a small extension of the programming language Clojure[4] which allows embedding of languages defined by context-free grammars within Clojure source files[5].

[3] SYNFUL LISP handles arbitrary context-free grammars with one restriction: parentheses should always be balanced.

[4] http://www.clojure.org

[5] The sources of SYNFUL LISP can be downloaded from http://www.cwi.nl/˜storm/synful-lisp.tar.gz.

```
(def grant #stm(
  events
    doorClosed  D1CL
    drawOpened  D2OP
    lightOn     L1ON
    doorOpened  D1OP
    panelClosed PNCL
  end
  ...
  state unlockedPanel
    panelClosed => idle
  end))
```

```
(stm/stm [
  (stm/events ([
     (stm/event "doorClosed"  "D1CL")
     (stm/event "drawOpened"  "D2OP")
     (stm/event "lightOn"     "L1ON")
     (stm/event "doorOpened"  "D1OP")
     (stm/event "panelClosed" "PNCL")]))
  ...
  (stm/state ("unlockedPanel" [
     (stm/trans "panelClosed" "idle")]))])
```

Fig. 2: Embedded state machine syntax (left) and the resulting AST (right)

The basic idea is very similar to ordinary Lisp *reader macros*, which allow the reader to be extended with arbitrary parsing code [2,15]. In SYNFUL LISP, the reader is extended with parsers derived from context-free grammars instead.

There are three steps in using the extension:

1. Define a context-free grammar for an embedded language; this is done using the embedded language for grammars.
2. Hook it into Clojure's Lisp reader with a grammar specific dispatch token τ, and a symbol corresponding to its semantic function or macro f.
3. Use the embedded syntax delimited by `#` τ(...).

Since the Clojure facilities of functions and macros are used to implement the semantics of embedded fragments, SYNFUL LISP can be classified as a homogeneous embedding approach [16].

An example of a context-free grammar for state machines is displayed as Clojure module in Figure 1[6]. The first line states that the following definitions are within the name space `stm`. Next, the `metagrammar` module is imported with the Clojure `use` directive. Finally, a (global) variable `*stm-grammar*` is defined to be equal to the value of the embedded grammar (delimited by `#grammar(` and `)`).

Grammar productions are defined using well-known grammatical constructs, such as `*` for iteration and `?` for optionality. Currently, the lexical structure of embedded languages is fixed: the grammar formalism has special symbols for terminal types identifier (`id`), numbers (`int`) and strings (`str`). Literals used in the grammar (e.g., `"state"`) are automatically reserved and excluded from `id` to avoid ambiguities. Productions can be labeled to indicate AST node types. For instance the non-terminal `Decls` is labeled `stm`.

The state machine syntax can be hooked into the Lisp reader with the following statement:

[6] This language is based on Martin Fowler's introductory example at http://martinfowler.com/dslwip/Intro.html.

```
(embed "stm" *stm-grammar* "Decls" "stm" 'quote)
```

This can be read as the following instruction to the reader: when you encounter "stm" after the dispatch character "#", start parsing the characters that follow (enclosed in parentheses) using *stm-grammar*, using "Decls" as start non-terminal. When parsing is finished, construct an AST the node types of which are in name space stm, and apply the function/macro/form quote to it. In this case, the semantic "function" (quote) just returns the original AST.

The state machine syntax is immediately usable after the embed statement. The left-hand side of Figure 2 shows an excerpt of embedding state machine syntax in Clojure. Because the semantic function passed to embed is quote, the variable grant is bound to the AST of the embedded fragment. The AST is shown on the right-hand side of the figure. Note how the forward slash (/) is used to qualify the symbols in the AST using the name space provided to embed.

Escaping to Clojure In SYNFUL LISP, it is possible to escape to ordinary Clojure code using the "'" (backtick) character[7]. Where an escape occurs, is indicated in the grammar using the special terminal symbol form.

To illustrate its use, consider changing the rule for Command in Figure 1 to Command ::= command: id form;. This means, that instead of using an identifier to identify the command code, this rule allows arbitrary Clojure code to occur after the command name. The following example simply prints out a log message when the unlockPanel is executed.

```
#stm(commands unlockPanel '(print "panel unlocked") end)
```

The corresponding AST is:

```
(stm/stm [(stm/commands ([(stm/command "unlockPanel"
   (print "panel unlocked"))]))])
```

An interpreter function for state machines could then use Clojure's eval to evaluate such Clojure fragments. On the other hand, a compiler macro could just leave the Clojure fragments in the expanded code.

3 Token Interception

Consider a typical parsing tool chain: a *tokenizer* converts a stream of characters into a stream of tokens. This stream of tokens is input to a *parser*, which in turn produces an abstract syntax tree (AST). A token *interceptor* sits in between the tokenizer and the parser; it wraps a tokenizer, and acts like a tokenizer to the parser.

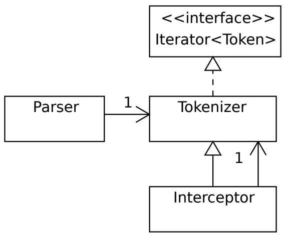


Fig. 3: Class diagram illustrating the token interception pattern

Figure 3 shows a class diagram highlighting the important relations in the pattern. Tokenizers are assumed to be a kind of Iterators that produce streams of tokens. Interceptors publish the same interface as tokenizers, but also wrap another, base tokenizer. Finally, parsers query tokenizers for the `next` token.

An interceptor acts as a filter: it may decide not to pass all tokens to the parser, but instead process some of them differently, for instance, by sending the tokens to a different parser. The result of this “child” parse can be provided to the original (“parent”) parser as ordinary tokens. Thus, one parser’s parse tree is another parser’s token.

A typical scenario illustrating token interception is displayed in the message sequence chart of Figure 4. In this diagram, tokenizers and interceptors are queried by parsers, via the `next` method. The `next` method is expected to return the next available token.

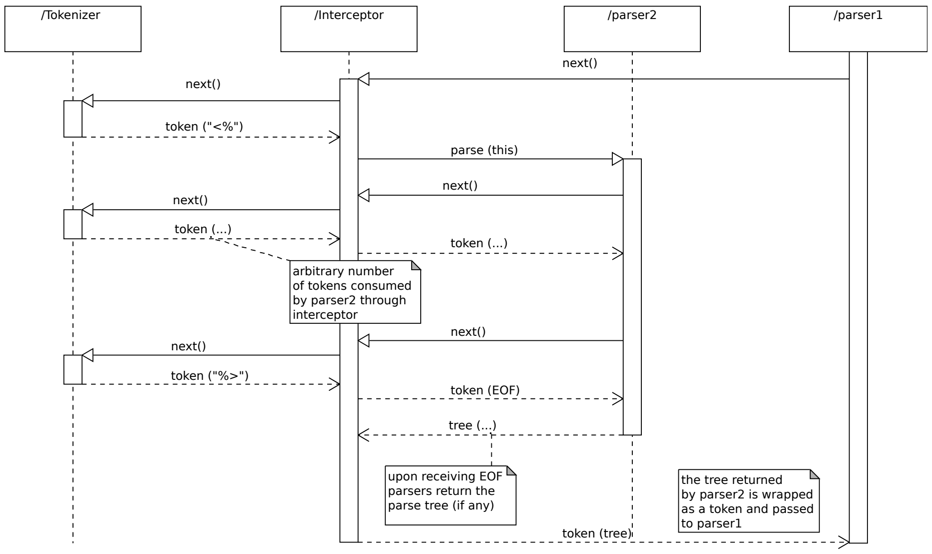
The figure shows four active objects: `Tokenizer`, `Interceptor`, `parser1` and `parser2`. `Parser1` is the currently active parser object, which asks the `Interceptor` object for the next token. The interceptor delegates this request to the `Tokenizer` object which does the actual tokenization; it analyzes the input and returns “token” objects.

Most of the time, the `Interceptor` will just return the tokens received from the `Tokenizer` to `parser1`. However, in the diagram, the token received from the tokenizer is a special opening quote `"<%"`[8]. Instead of returning the token to `parser1`, a different parser (`parser2`) is invoked through the `parse` method.

In this example, the `parser2` object will use the same tokenizer (in fact, interceptor), as was used by `parser1`, but this is not required. Again, most of the tokens `Interceptor` receives from `Tokenizer` will be returned directly to `parser2`. However, the special closing quote `"%>"` indicates that the embedded fragment that is currently being processed ends here. Therefore, upon intercepting this token, the `Interceptor` returns the end-of-file (EOF) token to `parser2` to indicate that all input has been consumed. As a result, the `parse` invocation will return control to the `Interceptor` with a parse tree. Finally, the `Interceptor` wraps this tree as token and provides it to `parser1` as the return value of the first `next` invocation in the diagram.

[7] As a result the character cannot be used as a literal in the grammars that are used for describing embedded languages.

[8] This is just an example; it could be anything.



/Tokenizer
/Interceptor
/parser2
/parser1
next()
next()
token ("<%")
parse (this)
next()
next()
token (...)
token (...)
arbitrary number of tokens consumed by parser2 through interceptor
next()
next()
token ("%>")
token (EOF)
tree (...)
upon receiving EOF parsers return the parse tree (if any)
the tree returned by parser2 is wrapped as a token and passed to parser1
token (tree)

Fig. 4: Message sequence chart illustrating the control flow of token interception

3.1 Multi-Level

The example of Figure 4 shows only one level of embedding. Token interception can easily accommodate multiple levels of embedding, including *escaping*: embedding the host language itself within the embedded fragment.

Multiple levels of embedding can be supporting by letting the interceptor maintain the nesting level of quotes and anti-quotes. Note that in the example, `"<%"` and `"%>"` are used to go from the host language to the embedded language; to go from embedded language to yet another embedding might be triggered by different quote tokens.

To escape from embedded language to host language can be catered for as well. This requires the interception of an anti-quote. If such a token is intercepted, the original parser is invoked (again), and the resulting tree is wrapped as a token for the second-level parser (e.g., `parser2`).

In the example, the second parser just uses the current interceptor as for its tokenizer. This does not have to be the case, of course. The `Interceptor` object could, for instance, instantiate a new interceptor object, wrapping `this`, `Tokenizer` or a completely different tokenizer. Embedded languages, therefore need not conform to the same lexical structure as the host language. The only requirement is that the closing quote (e.g., `"%>"`) can be detected.

3.2 On Demand

The token interception scheme as described in this paper, can be seen as an example of lazy, demand-driven or stream-like programming: an interceptor converts a stream of tokens to another stream of tokens. This happens lazily due to giving control to the

```
public EmbeddingInterceptor extends Interceptor {
  private int nesting = 1;

  public Token next() {
      Token t = tokenizer.next();
      if (t.is("(")) nesting++;
      if (t.is(")")) {
         nesting--;
         if (nesting == 0) return Token.EOF;
      }
      if (t.is("‘"))
         return new Form(LispReader.read(tokenizer.input));
      return t;
  }

  public boolean hasNext() {
     return nesting == 0 ? false : tokenizer.hasNext();
  }
}
```

Fig. 5: Embedding interceptor class

parsers at the end of the processing pipe line. Because the interceptors are the target of the `next` query of the parsers, they can decide to temporarily suspend an active parsing process, and resume them when appropriate.

The fact that the input is processed one token at the time, has the advantage that processing of earlier fragments of the input may have an effect on the way later parts of the input will be tokenized and/or parsed. For instance, a description of a grammar somewhere at the beginning of a file, can be effectively used to parse later parts of the same file.

4 Implementation of SYNFUL LISP

The Lisp reader of Clojure is a hand-written recursive descent parser using a number of dispatch tables to invoke the right parse function for a certain character. One of the tables used in the Clojure reader is the `dispatchMacros` table, which dispatches based on characters after the `#` character. This table has been modified so that for any alphabetic character after `#`, the reader branches to the routine which starts the grammar-based parsing, the `EmbeddingReader`.

The `EmbeddingReader` consumes all letters after the `#`-character to obtain the name of the grammar that should be used. This grammar is then looked up in a table. The tokenizer is instantiated for the current input stream, then wrapped in a token interceptor and passed to the parser derived from the grammar. The nature of token interception is discussed below. Currently, embedded fragments are parsed using a version of the GLR algorithm [11].

Component	Source lines of code (SLOC)
Parser generator + GLR	2617
Embedding	1245
Modifications to Clojure	4

Table 1: Source code statistics of Synful Lisp

The default tokenizer categorizes the input stream in token types string, literal, identifier, integer while discarding white-space and comments. These tokens are intercepted by a token interceptor similar to the one displayed in Figure 5. Finally, the GLR parser communicates to the interceptor instead of directly to the tokenizer.

The `next` method of the interceptor in Figure 5 returns the next token in the input stream. First the actual next token is assigned to `t` by delegating to the `next` method in the wrapped `tokenizer`. Then, if the retrieved token corresponds to the literal string `"("`, the level of `nesting` is increased by one. If, on the other hand, the token corresponds to a closing parenthesis, the `nesting` is decreased by one. Furthermore, if in the latter case `nesting` reaches 0, this means that the last, outermost closing parenthesis has been encountered. In other words, it means that the embedded fragment is stops here. The token itself is not passed to the parser. Instead the end-of-file token is sent to indicate that the complete string has been processed[9]. Otherwise, i.e. if `nesting` > 0, the token itself is returned in the last statement of the `next` method.

In order to support escaping to ordinary Clojure code within embedded syntax fragments, the tokenizer intercepts “`‘`“ (backtick) tokens, and branches back to the original Clojure `LispReader`. The resulting form is wrapped in a special `Form` token, which is handled by the parser as a special token, like `int`, `str`, and `id`.

4.1 Assessment

The implementation of syntax embedding using token interception in Clojure was achieved with a relative small amount of code. High-level size statistics in terms of source lines of code (SLOC) are displayed in Table 1. The largest component is the implementation of LR(0) parser generator and the GLR algorithm. This component is generic and is actually reused from a different project. The communication between the Clojure reader and the GLR parser, including the interception code is currently 1245 SLOC. Only 4 lines of original Clojure code had to be modified to make Synful Lisp.

The experiment suggests that token interception indeed is a simple and elegant solution for implementing syntax embedding in Clojure. The encouraging size statistics, however, require some qualification. The current implementation could be so small for a number of reasons:

[9] The astute reader will have noticed that this “trick” to detect the end-of-input only works if the embedded fragment itself does not contain any unbalanced parentheses. It is for this reason that grammars that feature productions with unbalanced parentheses are rejected. Since most languages only feature parentheses in a balanced way anyway, this is considered to be a small price to pay.

- Hooking into the Lisp reader is easy because it already uses dispatch tables.
- Escaping to Clojure is simple, since the Lisp reader only recognizes a single syntactic type (expressions) and does not require and end-of-input token.
- S-expressions are a natural fit for AST representation. As a result, uniform integration of the result of embedded fragments into the main program is trivial.

It is instructive to look one step ahead, reflect upon the experiment of this paper, and try to assess how token interception could be used to implement syntax embedding, in other languages, such as Java.

Assuming a tokenizer and parser for Java are available and both are in the shape required by the token interception pattern, one would proceed as follows:

- The Java tokenizer should be adapted to produce a special token, say`#`, which can be intercepted. The interceptor can then determine the grammar that should be used from the letters after `#` just as in SYNFUL LISP.
- The grammars used for embedded fragments may or may not use the same tokenizer that is used by the Java parser; this is a design decision. In both cases, the tokenizer should at least produce tokens for `(` and `)` and escape quotes.
- The escaping quote is parametrized in the non-terminal type that is allowed. For instance, if escaping to both statements and expressions is required, they must be triggered by different escape quotes. If the original Java parser requires an EOF token (as most bottom-up parsers do), the escape quote should be accompanied by a closing quote[10].
- The Java grammar/parser should be adapted to be able to deal with the result of parsing embedded fragments. For instance, special token types should be added to accept the wrapped trees for sub parses.

As far as semantics is concerned, this would require a lot of additional work in the back end of the Java processing pipe line. The token interception pattern, however, is only concerned with mid-stream switching of parsers, and does not affect the way parsed results are processed in later phases.

A more substantial evaluation would compare the features of SYNFUL LISP to other syntactic embedding approaches, while at the same estimating the complexity, size and flexibility of the implementation. Such an evaluation would be largely qualitative, since the trade-offs and design decisions made in different systems complicate comparison. I consider such an evaluation as an important direction for future work.

5 Concluding Remarks

Syntax embedding is an attractive pattern for the implementation of DSLs. It leverages context-free syntax for language definition, while at the same time providing tight integration with a general host purpose language. I have presented token interception as a simple pattern for implementing general, dynamic and multi-level syntax embeddings. The pattern has been used in the implementation of SYNFUL LISP, an extension of Clojure that supports embedded languages described by arbitrary context-free grammars.

[10] This was not needed in the case of Clojure, since the Lisp reader knows when it has finished parsing an expression.

References

1. Aasa, A., Petersson, K., Synek, D.: Concrete syntax for data objects in functional languages. In: Proceedings of the 1988 ACM conference on LISP and functional programming (LFP'88). pp. 96–105 (1988)
2. Baker, H.G.: Pragmatic parsing in Common Lisp; or, putting `defmacro` on steroids. SIGPLAN Lisp Pointers IV(2), 3–15 (1991)
3. Boullier, P.: Dynamic grammars and semantic analysis. Research Report RR-2322, INRIA (1994)
4. Brabrand, C., Schwartzbach, M.I.: The `metafront` system: Safe and extensible parsing and transformation. Science of Computer Programming Journal (SCP) 68(1), 2–20 (2007)
5. Brand, M., Deursen, A., Heering, J., Jong, H., Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P., Scheerder, J., Vinju, J., Visser, E., Visser, J.: The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In: Wilhelm, R. (ed.) Compiler Construction (CC '01). Lecture Notes in Computer Science, vol. 2027, pp. 365–370. Springer-Verlag (2001)
6. Bravenboer, M., Visser, E.: Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04). pp. 365–383. ACM (2004)
7. Cardelli, L., Matthes, F., Abadi, M.: Extensible grammars for language specialization. In: Beeri, C., Ohori, A., Shasha, D. (eds.) Proceedings of the Fourth International Workshop on Database Programming Languages – Object Models and Languages (DBPL-4). pp. 11–31. Workshops in Computing, Springer (1993)
8. Christiansen, H.: A survey of adaptable grammars. SIGPLAN Not. 25(11), 35–44 (1990)
9. Knöll, R., Mezini, M.: π: a pattern language. In: Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA'09). pp. 503–522. ACM (2009)
10. Moon, D.A.: Programming language for old timers. Online: http://users.rcn.com/david-moon/PLOT/ (April 2009)
11. Rekers, J.: Parser Generation for Interactive Environments. Ph.D. thesis, University of Amsterdam (1992)
12. Renggli, L., Denker, M., Nierstrasz, O.: Language boxes: Bending the host language with modular language changes. In: Proceedings of the 2nd International Conference on Software Language Engineering (SLE'09). LNCS, Springer (2009)
13. Seaton, C.: A programming language where the syntax and semantics are mutable at runtime. Tech. Rep. CSTR-07-005, University of Bristol (June 2007)
14. Spinellis, D.: Notable design patterns for domain-specific languages. Journal of Systems and Software 56(1), 91–99 (2001)
15. Steele, G.L.: Common Lisp the Language. Digital Press, 2nd edn. (1990)
16. Tratt, L.: Domain specific language implementation via compile-time meta-programming. ACM Trans. Program. Lang. Syst. 30(6) (2008)
17. Warth, A., Piumarta, I.: Ometa: an object-oriented language for pattern matching. In: Proceedings of the 2007 symposium on Dynamic languages (DLS'07). pp. 11–19. ACM (2007)