# Modular Syntax

(ICFP 2015 Student Research Competition)

Cyrus Omar
Carnegie Mellon University
comar@cs.cmu.edu

## Abstract

Practical functional programming languages like ML and Haskell often include derived syntactic forms that capture common idioms more concisely or naturally. For example, derived list syntax is built into all major functional languages. Many languages, and dialects thereof, go beyond this, building in derived syntax associated with other types of data, like vectors (SML/NJ), arrays (OCaml), monadic commands (Haskell), syntax trees (Scala, F#), XML (Scala, Ur/Web) and SQL queries (F#, Ur/Web).

We introduce a new functional language based on ML called Verse that takes a less *ad hoc* approach – rather than privileging particular library constructs with primitive syntactic support, Verse supports primitives that allow library providers to introduce new syntactic expansions on their own, in a safe, hygienic and modular manner.

## 1. Introduction

Let us begin from the perspective of a regular expression library provider, where the abstract syntax of patterns, $p$, over strings, $s$, is specified as below:

$$p ::= \textbf{empty} \mid \textbf{str}(s) \mid \textbf{seq}(p;p) \mid \textbf{or}(p;p) \mid \textbf{star}(p) \mid \textbf{group}(p)$$

In a language with an ML-style module system [7], one standard way to express this abstract syntax is by defining an *abstract datatype* with a signature like that shown in Figure 1. Clients of any module `P` that has been opaquely ascribed this signature, written `P :> PATTERN`, manipulate patterns as values of the type `P.t` using the interface described by this signature. The identity of this type is held abstract outside the module during typechecking (i.e. it acts as a newly generated type). This is beneficial in that it ensures that implementation details (e.g. internal compiled representations of patterns) do not escape (and thus the library provider can reason about them modularly and evolve them freely).

The abstract syntax of patterns is too verbose to be practical in all but the most trivial examples, so programmers conventionally write patterns using a more concise concrete syntax. For example, the concrete syntax `A|T|G|C` corresponds to the following much more verbose pattern expression:

```
P.Or(P.Str "A", P.Or(P.Str "T",
  P.Or(P.Str "G", P.Str "C")))
```

```
signature PATTERN = sig {
  type t
  val Empty : t
  val Str : string -> t
  val Seq : t * t -> t
  val Or : t * t -> t
  val Star : t -> t
  val Group : t -> t
  val case : (
    t ->
    'a -> (* Empty case *)
    (string -> 'a) -> (* Str case *)
    (t * t -> 'a) -> (* Seq case *)
    (t * t -> 'a) -> (* Or case *)
    (t -> 'a) -> (* Star case *)
    (t -> 'a) -> (* Group case *)
    'a)
}
```

**Figure 1.** A signature describing an abstract datatype.

## 2. Existing Approaches

### 2.1 Dynamic String Parsing

To expose this more concise concrete syntax for patterns to clients, the most common approach is to provide a function that parses strings to produce patterns. Because, as just mentioned, there may be many implementations of the `PATTERN` signature, the usual approach is to define a parameterized module (a.k.a. a *functor* in SML) defining utility functions like this abstractly:

```
module PatternUtil(P : PATTERN) => mod {
  fun parse(s : string) : P.t => (* ... *)
}
```

This allows a client of any module `P : PATTERN` to construct patterns like this:

```
let module PU = PatternUtil(P)
PU.parse "A|T|G|C"
```

Unfortunately, this approach is imperfect for several reasons:

1. The syntax of strings conflicts with the syntax for patterns (e.g. pattern escape sequences like `\d` are misparsed as invalid string escape sequences). Doubling backslashes (and other workarounds) increases syntactic cost.

2. Pattern parsing does not occur until a pattern is evaluated, so malformed patterns are not detected statically. This can also incur run-time cost, which must be managed carefully.

3. Idioms for constructing patterns compositionally are not captured by this technique (we will see an example below).

4. Using strings coincidentally to introduce patterns tempts programmers to use string concatenation in subtly incorrect

ways when working with string data. For example, consider the following function:

```
fun example_bad(name : string) =>
  pattern (name ^ ": \\d\\d\\d\\d")
```

This function works correctly for common inputs (i.e. alphabetic names), but when the input contains special characters that have meaning in the concrete syntax of patterns, a problem arises. In applications that query sensitive data, mistakes like this lead to *injection attacks*, which are among the most common and catastrophic security threads on the web today.

### 2.2 Direct Syntax Extension

One tempting alternative to dynamic string parsing is to use a system that gives the users of a language the power to directly extend its concrete syntax with new derived forms like Sugar* [2] or Camlp4 [6]. These allow us to directly introduce pattern syntax into our core language's grammar, perhaps following Unix conventions like this:

```
let val ssn = /\d\d\d-\d\d-\d\d\d\d/
```

For patterns constructed compositionally, we can define *splicing syntax* that allows us to write string and pattern expressions inline (distinguished by prefixes `@` and `%`, respectively). For example:

```
fun example_concise(name : string) =>
  /@name: %ssn/
```

Had we mistakenly written `%name`, we would encounter only a static type error, rather than the silent injection vulnerability discussed above.

The problem is that we cannot guarantee that *syntactic conflicts* between such extensions will not arise. If another library provider used overlapping syntax for a different implementation or variant of regular expressions, or for some other unrelated construct, then a client could not simultaneously use both libraries at the same time.

### 2.3 Macros

An alternative approach is to leave the concrete syntax of the language fixed, but repurpose it for novel ends using a *macro system*. The LISP macro system [4] is perhaps the most prominent example of such a system. In languages with a richer static type structure, variants of macros that restrict rewriting to a particular type and perform the rewriting statically have also been studied [3, 5] and integrated into languages, e.g. Scala [1].

For example, we might repurpose the infix operators that are normally used for other purposes to support string and pattern splicing as follows:

```
fun example_with_macro(name : string) =>
  pattern P (name ^ ": " + ssn)
```

Here, `pattern` is a macro parameterized by a module `P : PATTERN`. Having to creatively repurpose existing syntax in this way limits the impact a library provider can have on syntactic cost (particularly when it would be desirable to adopt conventions that are not consistent with the conventions adopted by the language). It also can create confusion for readers expecting parenthesized expressions to behave in a consistent manner.

## 3. Contributions

Verse provides a new external primitive – the **typed syntax macro** (TSM) – that combines the syntactic flexibility of syntax extensions with the reasoning guarantees of typed macros. TSMs can be parameterized by modules, so they can be used to define syntax valid at any type defined by a module satisfying a specified signature. This addresses all of the problems brought up above.

To introduce TSMs, let us consider the following concrete external expression:

```
pattern P /A|T|G|C/
```

Here, we apply a *parameterized TSM*, `pattern`, first to a module parameter, `P`, then to a *delimited form*, `/A|T|G|C/`. Note that a number of alternative delimiters are also provided by Verse's concrete syntax and could equivalently be used. The TSM statically parses the *body* of the provided delimited form, i.e. the characters between the delimiters (shown here in blue), and computes an *expansion*, i.e. another external expression. In this case, the expansion is the expression shown in Section 1.

The definition of `pattern` looks like this:

```
syntax pattern(P : PATTERN) at P.t {
  static fn (ps : ParseStream) : Exp =>
    (* pattern parser here *)
}
```

This TSM definition first identifies the TSM as `pattern`, then specifies a module parameter, `P`, which must match the signature `PATTERN`. Note that identifying the module parameter as `P` here is an entirely local choice, i.e. the TSM can be applied to *any* module parameter matching `PATTERN`. This parameter is used in the type annotation **at** `P.t`, which specifies that all elaborations that arise from the application of this TSM to a module `P` and a delimited form must necessarily be of type `P.t`. These elaborations arise by the static action of the parse function defined next, within braces. It is written as a static function of type `ParseStream -> Exp`. The type `ParseStream` will give the function access to the body of the delimited form (in blue above) and the sort `Exp` encodes the abstract syntax of external expressions.

To support splicing syntax as described in Sec. 2.2, the parse function must be able to extract external subexpressions directly from the parse stream. For example, consider the client code below:

```
(* TSMs can be partially applied and abbreviated *)
let syntax pat = pattern P
let val ssn = pat /\d\d\d-\d\d-\d\d\d\d/
fun example_tsm(name: string) =>
  pat /@name: %ssn/
```

The subexpressions `name` and `ssn` on the last line occur directly in the parse stream. When the parse function encounters these, it asks the `ParseStream` to extract these as *spliced expressions* for use in the elaboration. For example, the elaboration generated for the body of `example_tsm` above would, if written concretely with spliced expressions marked, be:

```
P.Seq(P.Str(<spliced>(name)),
  P.Seq(P.Str ": ", <spliced>(ssn)))
```

The hygiene mechanism then ensures that only these marked portions of the generated elaboration can refer to the variables at the use site, preventing inadvertent variable capture by the elaboration.

A formal specification of this mechanism is under development, but space limitations prevent us from introducing it here. In earlier work, we described a simpler variant of TSMs which do not support integration with an ML-style module system [8]. Module parameters represent the novel contribution of this submission.

## 4. Conclusion

Typed syntax macros allow library providers to define new syntactic elaborations, while guaranteeing to clients that these can be combined arbitrarily. They support a hygienic type discipline by the use of type annotations, and integration with Verse's ML-style module system by the use of module parameters. Taken together, we believe TSMs will decrease the need for syntactic dialects and supplant the use of *ad hoc* tools like Camlp4.

## References

[1] E. Burmako. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *Proceedings of the 4th Workshop on Scala (SCALA '13)*, pages 3:1–3:10, 2013.

[2] S. Erdweg and F. Rieger. A framework for extensible languages. In *GPCE '13*, pages 3–12, 2013.

[3] S. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *ICFP '01*, pages 74–85, 2001.

[4] T. P. Hart. MACRO definitions for LISP. Report A. I. MEMO 57, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, Oct. 1963.

[5] D. Herman. *A Theory of Typed Hygienic Macros*. PhD thesis, Northeastern University, Boston, MA, May 2010.

[6] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system release 4.01 Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, September 2013.

[7] D. MacQueen. Modules for Standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 198–207, 1984. ISBN 0-89791-142-3. doi: 10.1145/800055.802036. URL http://doi.acm.org/10.1145/800055.802036.

[8] C. Omar, C. Wang, and J. Aldrich. Composable and hygienic typed syntax macros. In *ACM Symposium on Applied Computing (SAC '15)*, 2015.