Whitespace-Delimited, Type-Directed Parsing for Embedded DSLs

Cyrus Omar, Benjamin Chung[1], Darya Kurilova, Alex Potanin[2], and Jonathan Aldrich
Carnegie Mellon University
{comar, darya, aldrich}@cs.cmu.edu, bwchung@andrew.cmu.edu[1], and alex@ecs.vuw.ac.nz[2]

Abstract

Domain-specific languages enhance ease of use, expressiveness and verifiability, but defining and using different DSLs within a single application remains difficult. We present an approach to embedded DSLs where: 1) a whitespace-delimited lexer is shared across DSLs, and 2) the parsing and type checking phases occur in tandem so that types can define domain-specific parsing strategies governing whitespace-delimited blocks. We argue that this approach occupies a sweet spot with both high expressiveness and ease of use. We outline a design and ongoing implementation of this strategy and examine the relationship between the type-directed and keyword-directed strategies for parsing of embedded DSLs.

1. Introduction

Domain-specific languages (DSLs) [4, 8, 11] enable the expression of domain-specific ideas naturally and permit domain-specific verification and compilation strategies. For DSLs to reach their full potential, however, it must be possible to easily use multiple DSLs at once within a host general-purpose language (GPL), such that DSL code and GPL glue code can inter-operate to form a complete application. More specifically, we propose the following essential design criteria:

- *Composability*: It should be possible to use multiple DSLs and a GPL within a single program unit. Within the text file-based paradigm familiar to most developers, this means including multiple DSLs within a single file. Moreover, it should be possible to embed code written in one DSL within another DSL without them requiring knowledge of each other or interfering with one another. So, more specifically, DSLs should be *safely composable*.
- *Interoperability*: DSLs should be able to *interoperate*. A necessary condition for interoperability is that it should be possible to pass and operate on values defined in foreign DSLs. Additional requirements, such as the ability to do so naturally (rather than requiring a large amount of "glue code") and safely may also be relevant in many settings. Minimally, however, if one DSL defines a function or data structure satisfying an interface specified in a common interface description language (such as the type system of a host GPL), code written in another DSL should be able to use those values according to that interface, without requiring that the client DSL have knowledge of the details of the provider DSL or *vice versa*.

In addition to composability and interoperability, we believe that to be most useful, a system supporting DSLs should have the following characteristics:

- *Flexibility*: Have flexible enough DSL syntax to support a wide variety of text-based notations;
- *Identifiability*: Make it easy for programmers to identify which code is written in which DSL;
- *Consistency*: Encourage DSLs written in similar styles in order to enhance readability and learnability of each DSL;
- *Simplicity*: Keep the cost of defining a DSL low.

We are developing a comprehensive approach that has the potential to meet these design requirements well, based on defining DSLs within a language specifically designed for extensibility from within. The approach inherently supports multiple DSLs in the same file, and supports interoperability because the DSLs are defined by translation into a common internal language. One key aspect of our proposed solution, and one of the focal points of this paper, is that we restrict the syntax of DSLs in a few key ways that greatly simplify the mechanism without significantly decreasing expressiveness:

- The host language and its DSLs share a tokenization and lexing strategy, standardizing conventions for identifiers, operators, constants, and comments. This avoids the cost of defining lexing within each DSL, avoids many kinds of low-level clashes between languages, and makes the composed language more readable. Note that this does not limit the ability of a DSL to define new keywords and other constructs.
- Constructs in the host language and its DSLs are *whitespace-delimited* at the top level, according to a particular strategy that we will describe. Various forms of parentheses can also serve as delimiters. Thus, indentation levels or parenthesized expressions clearly delimit blocks that are governed by a particular language. This makes the boundaries of each DSL clear to the programmer and the compiler, enhancing usability and guaranteeing that subtle conflicts cannot arise.

```
1  val hotProductDashboard = new Application
2     external component twitter : Feed
3        location [www.twitter.com]
4     external component client : Browser
5        connects to servlet
6     component servlet : DashServlet
7        connects to productDB, twitter
8        location [intranet.nameless.com]
9     component productDB : Database
10       location [db.nameless.com]
11    policy NamelessSecurity
12       must salt [servlet.login.password]
13       connect * -> servlet with HTTPS
14       connect servlet -> productDB with TLS
```

Figure 1. Wyvern DSL: Architecture Specification

```
1  val newProds = productDB.query
2     select twHandle
3     where introduced - today < 3 months
4  this.prodTwt = Feed(newProds)
5  return prodTwt.query
6     select *
7     group by followed
8     where count > POPULAR
```

Figure 2. Wyvern DSL: Queries

Within this syntactic framework, we then propose a novel syntax extension mechanism in which the *expected type of an expression*, rather than an explicit keyword, signals the switch to a new DSL that parses the tokenized and lexed code within a subsequent delimited block according to a type-associated grammar. The more common keyword-directed strategy arises as a special case of this type-directed strategy. We will begin with some motivating examples (§2), describe our approach (§3) and implementation (§4), and conclude with related work (§5).

2. Motivating Examples

We start with a few examples meant to illustrate the expressiveness of our approach and the breadth of DSLs we plan for it to support. The examples are presented in the proposed syntax for Wyvern, a new language being developed by our group initially targeted toward building secure web and mobile applications. We will further clarify how these examples are parsed in Section 3.

The first example, shown in Figure 1, describes the overall architecture of a "hot product dashboard" application. The constructor `Application` takes an argument of type `ComponentArch`. Rather than being provided explicitly, a DSL for specifying the component architecture of the application is used to provide this argument because it appears in a whitespace-delimited block. The example architecture declares several components, some of which are marked **`external`** to indicate that they are used by this application but are not part of it directly. Component types, connectivity and properties, such as location, are declared within further-indented blocks.

The **`policy`** keyword introduces a security policy, which constrains the communication protocols to be used as well as enforces secure handling of passwords. Again, indentation delimits the scope of the security policy. This illustrates that one DSL may be nested within another.

```
1  html:
2     head:
3        title: [text "Hot Products"]
4        style:
5           myStylesheet
6     body:
7        div id="search":
8           SearchBox("products")
9        div id="products":
10          FeedBox(servlet.hotProds())
```

Figure 3. Wyvern DSL: Presentation

```
1  class ComponentArch
2   grammar ::= ((component|policy) \n)+ where
3     component ::=
4       ("external ")? "component " ID ":" TYPE
5         [componentProperty*]?
6     componentProperty ::=
7        "location "(EXP of URL)
8       | "connects to" (ID ",")* ID
9     ...
```

Figure 4. Type-Associated Grammar Definition

Figure 2 shows how a DSL for querying can be used from within ordinary Wyvern code. The example shows the body of a method for computing a feed that is derived from tweets about the company's new products. In this example, the use of a querying DSL is triggered by the use of a function, query, expecting a `DBQuery`. This type defines a particular syntax for database queries, delimited by the indented block as the reader would expect. This is similar to what can be expressed with built-in query syntax such as Microsoft's LINQ [3], but in this case, defined in a user-defined type. The feed query operator may define different features, as seen on Line 5.

Finally, Figure 3 shows a DSL for rendering the hot product application in a web browser. The DSL is based on HTML but uses our indentation-based syntax and allows Wyvern code to be integrated – function/method calls and variables of the appropriate type can be used at any point. Literal tags are introduced by a tag name followed by any tag attributes followed by a colon.

3. Approach

Figure 4 illustrates our approach to type-based extension for the example in Figure 1. The class `ComponentArch` defines a grammar extension to the language's expression grammar. This grammar must not conflict with the fixed base expression grammar, but because it is only used when an expression of the type `ComponentArch` type is requested *and* a whitespace-delimited block has been introduced, there is no need to guarantee that this grammar does not interfere with other type-associated grammars.

The declared grammar uses a standard BNF notation except for the declaration `EXP of URL` on Line 7. This indicates that at this position in the grammar, a term of type `URL` is needed. Thus, the domain-specific grammar (if any) associated with the `URL` class may be used here. Because URLs are typically short, it may be undesirable to require a new whitespace-delimited block, so the grammar defines blocks delimited by brackets as equivalent to whitespace-delimited blocks as well, as can be seen being used in Figure 1.

Parsing and Typechecking Wyvern source code is parsed using a standard whitespace-delimited lexer followed by an initial typechecking and parsing phase that occurs in tandem. The lexer for Wyvern generates a stream of tokens structured hierarchically, where indentation (and the equivalent bracket notation) increases the depth of the stream. Indentation and de-indentation can thus be simply understood as a special pair of tokens, `INDENT` and `DEDENT`, generated by the lexer and delimiting portions of the stream that will be handled in a special way during the next phase.

The standard Wyvern parser reads at a single level of the stream at a time. Initially, it parses the stream as a term of sort `Exp`, which represents the core expressions and declaration forms of the language. This sort includes basic facilities for declaring and working with functions and variables, objects and perhaps other to-be-determined core data types. When an indented block is encountered, however, it is left in-place within the parsed syntax tree.

When the parsing phase at a particular level is complete, it is typechecked according to a largely conventional protocol. In Wyvern, the type of a function is determined by its declaration, and cannot be affected by its use (this constraint may be relaxed in some instances in future iterations of the language). Thus, when a function is being applied, the expected types of its arguments are known. If, at a particular location where a function argument was expected, an indented block is found, then it is parsed not according to the `Exp` grammar, but according to the *type-associated grammar* contained in that type's definition. Figure 4 shows an example of such a grammar. That grammar may contain productions of the form `EXP of` τ where τ is another type in scope. A delimited block must be present at this location as well, and it will be parsed by the type-associated grammar of τ.

A particular function application can contain only one *primary* indented block. If multiple arguments support DSL syntax, then all those other than the first in the argument ordering must be invoked by a partially-indented keyword. This can be seen in Figure 5. Note that this partially-indented keyword ends the first delimited block and begins a new block for the keyword argument `toServer` of the function `db.sendQuery`. This can be seen as similar in form to keywords in functions in Smalltalk [9].

Keyword-Directed Invocation Keyword-directed invocation of a DSL is simply a special case of this approach. In particular, a keyword can be defined as a function with a single argument of a type specific to that keyword. The type contains the implementation of the domain-specific syntax associated with that keyword. If the construction of this type is controlled such that it can only be via the domain-specific syntax (and not any equivalent syntax in the core language), then it is precisely equivalent to keyword-directed invocation of a DSL.

As keyword-directed DSL invocation is highly explicit, we allow keywords to appear not just at the end of a line, but also within expressions. In this case, the keyword determines how much of the expression (and potentially following indented lines) that it will parse.

Procedural Parsing The grammar definition in Figure 4 is declarative and relies on a simple parser generation framework included in Wyvern. It may be desirable for more sophisticated parsing algorithms to be encoded for particular domain-specific languages. For example, an interoperability layer with another full-scale programming language may wish to support parsing features not expressible in Wyvern's declarative grammar language. It can be observed, however, that a declarative grammar inside a class definition can be thought of as inducing a class method (a concept also borrowed from Smalltalk [9]) called `parse`, that transforms a stream of tokens to an AST representation. This interface can be exposed directly to programmers who wish to extend the parsing capabilities of the language in more sophisticated ways. We leave the details of the AST representation and its processing during later stages of compilation as future work.

```
1 db.sendQuery
2     select x where x > 10
3   toServer
4     www.server.org
```

Figure 5. Function applications involving multiple whitespace delimited arguments use partially-indented keywords.

4. Implementation Considerations

The Wyvern[1] compiler was developed from the ground up to support the extensible parsing interface. The compiler is currently implemented in Java, with an eventual goal of self-hosting it within Wyvern itself.

Wyvern uses a fixed whitespace-indented lexing approach similar to that used by languages like Python. The current indentation convention is based on the number of whitespace characters. In order for a block to be formed, each of its lines must share the same initial whitespace string. If the indentation level is decreased but does not return to the initial indentation level, a keyword block is created as shown in Figure 5. This particular approach is not found in other whitespace-delimited languages to our knowledge.

The Wyvern front-end effectively combines the parsing and type checking stages that are usually separated in more traditional compilers, such as `javac` or `gcc`. Before any block can be parsed, Wyvern requires a context that contains the types of the arguments of the surrounding function and any variables and types in scope. This context is passed into the delimited block for portions of the type-associated grammar that contain terms of sort `Exp` or `Type`, and thus may contain other function calls, variables, or types.

4.1 Discussion and Future Work

Custom Contexts Presently, grammars cannot define their own contexts that are passed through nested grammars, but this would likely be a useful addition in many cases. As an example, consider a **`let`** expression defined by a functional programming based DSL in Wyvern:

```
1 let
2     x = 5
3     z = 4
4   in
5     x + z
```

This expression requires maintaining a context of bound variables. This context must be maintained such that any expression within the `in` block that is written in this DSL has access to the appropriate context, even if it appears as a subexpression within a different DSL. Multiple DSLs may define different contexts, and these must be threaded throughout nested DSLs in a modular manner.

[1] `https://github.com/wyvernlang/wyvern`

Custom Lexers Our existing lexing strategy may be too restrictive, requiring all DSLs to be hierarchical in nature. One potential expansion would be to enable DSLs to define their own lexers, still perhaps delimited by indentation or parentheses. Such an extension would sacrifice some readability.

Wyvern's operators are built in and their precedence follows that of C++ operators. We do not allow a replacement parser for infix operators as we considered it to unnecessarily complicate the current prefixed parsing approach. In the future, we plan to further support redefining operators.

Finally, following Python and some other whitespace delimited languages, we may wish to allow parenthesized expressions to avoid the need for following the indentation level. This is still subject to debate and, as we try to express more and more DSL kinds in Wyvern, we may decide to enforce indentation levels even inside the parentheses.

5. Related Work

The most common mechanism for creating programming language extensions is macros, as exemplified by hygienic macros in Scheme. Macros in Scheme and other Lisp-style languages are written in Lisp itself, and benefit from the regular syntax of the underlying language and the consistent use of parentheses as expression delimiters. The syntactic nature of macros allows them to control parsing within a delimited scope, and this delimited control of parsing inspired our approach in Wyvern. Wyvern's use of types to trigger custom parsers avoids the Lisp limitation of starting each DSL with a keyword or macro name.

Wyvern chooses whitespace delimitation rather than parenthesis delimitation because, anecdotally, many programmers find syntax in that style to be more readable. There are several proposals for making Lisp and its dialects whitespace-delimited, including sweet-expressions [19] and SRFI-49 [14] for Scheme, P4P proposal for Racket [12], Kernel's f-expressions [16] for Common Lisp [17], and others. None of them seem to be successfully implemented and deployed within the Lisp communities, however. This may be because Lisp programmers don't want to switch away from parentheses, yet having an alternate syntax is not by itself enough to draw non-Lisp programmers into the community.

Some language extensibility projects provide extension at levels of abstraction above parsing. For instance, OJ (previously, OpenJava) [18] provides a macro system based on a meta-object protocol, and Backstage Java [15] employs compile-time meta-programming. Both of these systems provide macro-style rewriting of existing source code, but they provide at most limited extension of language parsing.

Other systems aim at providing more flexible forms of syntax extension compared to our whitespace-delimited approach, at the potential cost of complexity or conflicts. Camlp4 [6] is a preprocessor for OCaml that offers the developer the ability to extend the concrete syntax of the language via the use of parsers and extensible grammars. SugarJ [7] takes a library-centric approach which allows to syntactically extend the Java language by adding libraries. In Wyvern, the core language (that is, the `Exp` sort) does not get extended directly, so conflicts cannot arise and reasoning about which DSL is applicable is much simpler.

Finally, researchers have developed a number of DSL frameworks and workbenches, such as the Meta Programming System (MPS) [2], Spoofax [10], the Intentional Domain Workbench (IDW) [1], and Ensō [5], that provide support for extending a variety of programming languages by applying generic rules and concepts. The Marco language [13] similarly provides macro definition at a level of abstraction that is largely independent of the target language.

Compared to these approaches, Wyvern focuses on making one programming language extensible with embedded DSLs, provides a common lexer and extensible parsing confined to whitespace-delimited regions, and investigates a novel type-based approach to triggering parser extensions.

6. Conclusion

In this paper, we described how extensible parsing in Wyvern makes for a solid platform to support whitespace-delimited, type-directed embedded DSLs. In the future, we aim to implement a wide variety of DSLs in Wyvern tweaking our approach and implementation thereof to provide a comprehensive example of supporting multiple interacting DSLs in a safe and easy-to-use manner.

References

[1] Intentional Domain Workbench. `http://www.intentsoft.com/intentional-technology/`.

[2] JetBrains MPS – Meta Programming System. `http://www.jetbrains.com/mps/`.

[3] LINQ (Language-Integrated Query). `http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx`.

[4] Adobe Systems Inc. *ColdFusion Developer's Guide*.

[5] William R. Cook, Alex Loh, and Tijs van der Storm. Ensō: A self-describing DSL workbench. `http://enso-lang.org/`.

[6] Daniel de Rauglaudre. *Camlp4 - Reference Manual*, Sep 2003.

[7] Sebastian Erdweg, Lennart C.L. Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser. Sugarj: library-based language extensibility. In *OOPSLA*, pages 187–188. ACM, 2011.

[8] Alan Harris and Konstantin Haase. *Sinatra: Up and Running*. O'Reilly Media, California, 1st edition, November 2011.

[9] Harry H. Porter III. Smalltalk: A White Paper Overview. March 2003.

[10] Lennart C. L. Kats and Eelco Visser. The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs. In *OOPSLA*, pages 444–463. ACM, 2010.

[11] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*, chapter 3. Using the Shell. Prentice Hall, 1984.

[12] Shriram Krishnamurthi. P4P: A Syntax Proposal. `http://shriram.github.com/p4p/`, 2010.

[13] Byeongcheol Lee, Robert Grimm, Martin Hirzel, and Kathryn S. McKinley. Marco: Safe, expressive macros for any language. In *ECOOP*, volume LNCS 7313, pages 356–382. Springer, 2012.

[14] Egil Möller. SRFI-49: Indentation-sensitive syntax. `http://srfi.schemers.org/srfi-49/srfi-49.html`, 2005.

[15] Zachary Palmer and Scott F. Smith. Backstage Java: Making a Difference in Metaprogramming. In *OOPSLA '11*, pages 939–958, New York, NY, USA, 2011. ACM.

[16] John N. Shutt. *Fexprs as the basis of Lisp function application or $vau: the ultimate abstraction*. PhD thesis, Worcester Polytechnic Institute, August 2010.

[17] Manuel Simoni. The meaning of forms in Lisp and Kernel. `http://axisofeval.blogspot.com/2011/07/meaning-of-forms.html`, 2011.

[18] Michiaki Tatsubori, Shigeru Chiba, M.-O. Killijian, and Kozo Itano. OpenJava: A Class-based Macro System for Java. In *Reflection and Software Engineering*, pages 117–133. Springer, 2000.

[19] David A. Wheeler. Readable Lisp S-expressions. `http://readable.sourceforge.net`, 2012.