Posted August 30, 2010 by [Eric Niebler](http://ericniebler.com), under [Boost](http://cpp-next.com/archive/category/boost/)  
From <http://cpp-next.com/archive/2010/08/expressive-c-introduction/>

**Expressive C++: Introduction**

This entry is part of a series, [Expressive C++»](http://cpp-next.com/archive/2010/08/expressive-c-introduction/) Entries in this series:

1. Expressive C++: ***Introduction***
2. [Expressive C++: Playing with Syntax](http://cpp-next.com/archive/2010/09/expressive-c-playing-with-syntax/)
3. [Expressive C++: Why Template Errors Suck and What You Can Do About It](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/)
4. [Expressive C++: A Lambda Library in 30 Lines (Part 1 of 2)](http://cpp-next.com/archive/2010/10/expressive-c-expression-extension-part-one/)
5. [Expressive C++: A Lambda Library in 30 Lines (Part 2 of 2)](http://cpp-next.com/archive/2010/10/expressive-c-expression-extension-part-two/)
6. [Expressive C++: Fun With Function Composition](http://cpp-next.com/archive/2010/11/expressive-c-fun-with-function-composition/)
7. [Expressive C++: Trouble With Tuples](http://cpp-next.com/archive/2010/11/expressive-c-trouble-with-tuples/)
8. [Expressive C++: Expression Optimization](http://cpp-next.com/archive/2011/01/expressive-c-expression-optimization/)

Hidden in C++ lurks[[1]](#footnote-1) another language—countless other languages, in fact—all of which are better than C++ at solving some types of problems. These *domain-specific* languages, be they languages for linear algebra, relational query or what-have-you, may only do one thing, but they do it well. We can create and use these other languages *directly within C++ itself*, using the power and flexibility of C++ to cut away[[2]](#footnote-2) the general-purpose parts of C++ we don’t want and replacing them with the domain-specific parts we do[[3]](#footnote-3). In this series of articles, we’ll be taking a close look at[[4]](#footnote-4) domain-specific languages, what they’re good for[[5]](#footnote-5), and how they can be easily implemented in C++ (with emphasis on *easily*) with the help of Boost.Proto.

**Domain-Specific Languages**

Back in 1958, [John Backus](http://en.wikipedia.org/wiki/John_W._Backus) was working on a vexing problem[[6]](#footnote-6): how to precisely nail down[[7]](#footnote-7) the syntax of the ALGOL programming language. What he came up with[[8]](#footnote-8) was a system for describing the syntax of ***any*** programming language. That system, much expanded over the years, is now known as [EBNF](http://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form), and it turns up in various guises[[9]](#footnote-9) wherever a computer language needs to be defined: official specifications, compiler-construction toolkits, wherever.

EBNF’s ubiquity is a testament to its simple, elegant power[[10]](#footnote-10). With just a few lines we can, for instance, describe the syntax, precedence, and associativity of the infix arithmetic operators:

expression = term ( ( ‘+’ term ) | ( ‘-’ term ) )\* ;   
term = factor ( ( ‘\*’ factor ) | ( ‘/’ factor ) )\* ;   
factor = UINT | ‘(‘ expression ‘)’ | ‘-’ factor ;

EBNF is a prime example of a domain-specific language; just the ticket for anyone[[11]](#footnote-11) dealing with syntax and parsing, but pretty useless for everybody else. The fact that it’s highly specialized is part of its appeal[[12]](#footnote-12). It lets you deal directly with the concepts relevant to the task at hand without distracting you with superfluous fluff[[13]](#footnote-13). There’s a downside though: if you need to do something that falls *just* outside of EBNF’s purview[[14]](#footnote-14), then EBNF is useless. This is the bugaboo of all domain-specific languages[[15]](#footnote-15): you often need a back-door so you can escape the confines of your domain when you have to[[16]](#footnote-16).

In contrast, C++ is a general-purpose programming language. Anything that can be done at all can be done with C++[[17]](#footnote-17). That power and flexibility comes with a cost[[18]](#footnote-18): there are many things for which C++ isn’t particularly great out of the box[[19]](#footnote-19). Compiler construction comes to mind[[20]](#footnote-20); C++ doesn’t directly support anything like EBNF. C++’s answer is to be great at defining libraries[[21]](#footnote-21). The onus then is on library writers to provide[[22]](#footnote-22) great domain-specific programming experiences. And by putting the domain-specific knowledge in a C++ library, users don’t fall off a cliff[[23]](#footnote-23) when they need to do something just outside of that library’s scope; they can always fall back on good ol’ C++[[24]](#footnote-24).

**A Language in a Library**

But what’s the best way to provide C++ library users with the domain-level notation and concepts with which they’re already familiar? “Modern” C++ library writers have come up with a creative solution[[25]](#footnote-25): use C++ expressions to approximate familiar domain-specific notation. Because they look like little languages built from parts of a bigger language, these libraries are called domain-specific *embedded* languages, or DSELs. If it’s done right, the results can really be quite impressive.Take, for instance, the [Boost.Spirit](http://boost-spirit.com) library, a parser generator library that gives the user a EBNF-like language with which to define parsers. Here is the Spirit equivalent to the above EBNF grammar:

/////////////////////////////////////////////////////////////////////

// A Spirit grammar that recognizes infix calculator expressions

/////////////////////////////////////////////////////////////////////

// boost::spirit::qi is where Boost.Spirit's parser generator lives.

**boost::spirit::qi::rule<Iterator> expression, term, factor;**

// An expression is a term followed by zero or more

// other terms separated by '+' or '-'

/\* The EBNF equivalent is:

expression = term ( ( '+' term ) | ( '-' term ) )\* ; \*/

**expression = term >> \*( ( '+' >> term ) | ( '-' >> term ) ) ;**

// A term is a factor followed by zero or more

// other factors separated by '\*' or '/'

/\* The EBNF equivalent is:

term = factor ( ( '\*' factor ) | ( '/' factor ) )\* ; \*/

**term = factor >> \*( ( '\*' >> factor ) | ( '/' >> factor ) ) ;**

// A factor is an unsigned integer or

// a parenthesized expression or

// a negated factor

**using boost::spirit::qi::uint\_;**

/\* The EBNF equivalent is:

factor = UINT | '(' expression ')' | '-' factor ; \*/

**factor = uint\_ | '(' >> expression >> ')' | '-' >> factor ;**

If you squint a bit[[26]](#footnote-26), you can see how the Spirit grammar corresponds to the EBNF. The correspondence isn’t perfect, but it captures the flavor[[27]](#footnote-27).[[28]](#footnote-28)

This example illustrates both the strength and the weakness of C++ expression-based DSELs. They are very good at getting you 90% of the way to your ideal syntax, but since your DSEL is limited to the operators [Brian Kernighan](http://en.wikipedia.org/wiki/Brian_Kernighan) and [Dennis Ritchie](http://en.wikipedia.org/wiki/Dennis_Ritchie) saw fit28-1 to give us, that remaining 10% can require us to take some creative license. Some take the 90% and are happy. Others find getting 90% of the way there more frustrating than simply taking a different route. And that’s OK. This article series isn’t for those people. No hard feelings28-2

A **rule** in Spirit is the C++ equivalent of a grammar rule in EBNF; the operators >> and | and \* have all been overloaded and are taken by Spirit to mean grammar sequencing, alternation and repetition, respectively. How does that work? The obvious (and sub-optimal) way would be for Spirit to make the right-shift operator do “right-shifting” and eagerly return a rule object. Instead, Spirit makes the right-shift operator return some dummy right-shift temporary object that simply holds on to its left and right operands. Likewise for all the other operators. The result is that an expression like:

factor >> \*( ( '\*' >> factor ) | ( '/' >> factor ) )

captures the whole expression—types, structure, values, and all—in an expression tree. When that tree gets assigned to a rule object, Spirit walks the tree and interprets it as a grammar rule. When all is said and done, the variable expression above holds a parser we can use to validate infix arithmetic expressions. Nice.

This is the general approach taken by expression-based DSELs: use operator overloading to build trees that capture the expression. Only *after* the whole expression is captured do you actually do any work. At that point, the tree can be analyzed, optimized, transformed and evaluated in domain-specific ways.

Spirit is a so-called compiler-construction toolkit like [yacc](http://dinosaur.compilertools.net). In addition to EBNF grammars, it provides other abstractions that compiler-construction experts have come to expect, like:

* [**Abstract syntax trees**](http://en.wikipedia.org/wiki/Abstract_syntax_tree): a standard tree representation of a parsed program,
* **Semantic actions**[**1**](http://cpp-next.com/archive/2010/08/expressive-c-introduction/1): bits of executable code you can attach to grammar rules that make your parser do stuff, and
* A host of **compiler algorithms** that operate on ASTs.

… all of which can be expressed directly in your C++ program as expressions. These basic abstractions—grammars, ASTs, semantic actions, and compiler algorithms—all become important later in a different context. Read on.

**This is the essence of it: by rigging things so that domain-level concepts can be expressed directly and concisely with C++ expressions, we can make users of our library comfortable and productive**[[29]](#footnote-29).

What’s more, we can often do it while realizing huge wins in performance[[30]](#footnote-30). With full access to a complete expression tree, a domain-specific library can analyze the tree and come up withsee 25 optimal evaluation strategies. A good example is [Blitz++](http://oonumerics.org/blitz/), a matrix crunching library[[31]](#footnote-31) that bested FORTRAN[[32]](#footnote-32), something that wasn’t thought possible at the time.[**2**](http://cpp-next.com/archive/2010/08/expressive-c-introduction/2) It achieves its remarkable performance by analyzing a *complete* expression tree and applying complex, domain-specific transformations like [loop fusion](http://en.wikipedia.org/wiki/Loop_fusion). That’s only possible because Blitz++ has access to the complete expression tree.

**Compiler Construction Toolkits**

The term is “domain-specific embedded *language*“, but you could be forgiven for thinking of Spirit quite simply as a “library”[[33]](#footnote-33). Imagine what would change if we all really believed to our core that[[34]](#footnote-34) DSELs actually *were* languages. Follow this logic: the libraries that implement DSELs would be like compilers, and *writing* such a library would be like writing a compiler. And as we’ve been discussing, writing compilers is a common enough task that tools like yacc and Spirit exist to make the job easier[[35]](#footnote-35). It follows that there should be a tool for DSEL writers that provides the sorts of abstractions compiler writers care about: grammars, ASTs, semantic actions, and compiler algorithms. What is needed is a compiler-construction toolkit for DSELs. [Boost.Proto](http://www.boost.org/libs/proto) is that toolkit.

An analogy with Spirit is apt[[36]](#footnote-36) since Spirit and Proto do pretty much the same thing, only at different times. Spirit generates programs that parse text at runtime. Proto generates programs that parse C++ expression trees at compile time.[**3**](http://cpp-next.com/archive/2010/08/expressive-c-introduction/3) Your run-of-the-mill[[37]](#footnote-37) compiler might be implemented with Spirit. In turn, Spirit—an expression-based DSEL—might be implemented with Proto, and in fact is.

I haven’t discussed how libraries like Spirit are typically implemented today, but suffice it to say that it involves a fair bit of[[38]](#footnote-38) [template metaprogramming](http://en.wikipedia.org/wiki/Template_metaprogramming), a notoriously difficult task[[39]](#footnote-39). Proto lets us write the Spirit library in terms of EBNF-like grammars and semantic actions, not the impenetrable and brittle[[40]](#footnote-40) template spaghetti code that is so often associated with template meta-programs. This would be a **big** step forward:

1. DSELs would be easier to develop and maintain;
2. DSELs would be rigorously and robustly[[41]](#footnote-41) defined, leading to friendlier interfaces and better error messages; and most importantly,
3. Code transformations that were prohibitively complex before would now be tractable.

The potential for complex code transformation opens endless possibilities. Can you see the possibilities yet? With such a toolkit, we could rip apart expressions, analyze the parts, and reassemble them[[42]](#footnote-42) to, say, do loop fusion for matrix calculations, or define our own syntax for transactions, or interpret an expression as a relational query, or … ??!!

**Hello, Boost.Proto**

Boost.Proto has been in [Boost](http://www.boost.org) since version 1.37. Since it’s a compiler-construction toolkit in the tradition of yacc and Spirit, Proto can build an AST for you. In Proto it’s as simple as this:

**#include <iostream>**

**#include <boost/proto/proto.hpp>**

**namespace proto = boost::proto;**

**int main()**

**{**

// Create a Proto terminal that wraps a string.

// Let's be cheeky and call it "cout".

**proto::terminal< char const \* >::type cout = { "cout" };**

// Create an expression tree and pass it to display\_expr

// for pretty-printing.

**proto::display\_expr(**

**cout << "hello" << ' ' << "proto!"**

**);**

**}**

The argument to proto::display\_expr isn’t an output expression, although we cheekily made it look like one[[43]](#footnote-43); instead, the expression builds a tree that *represents* the expression. What does it mean to left-shift strings and characters? *It doesn’t matter.* It’s just a tree that can be evaluated later, any number of times and any number of ways. The << operator in the expression is provided by Proto, along with every other operator. Building an AST with Proto is simple. You don’t have to define a single template or implement a single operator overload.

The result of running the above program is below:

shift\_left(

shift\_left(

shift\_left(

terminal(cout)

, terminal(hello)

)

, terminal( )

)

, terminal(proto!)

)

You can clearly see that Proto built a tree for us whose structure corresponds to the precedence and associativity of the C++ operators.

**What’s To Come[[44]](#footnote-44)**

As you might suspect[[45]](#footnote-45), Proto comes with tools for[[46]](#footnote-46) defining grammars that match expressions, and semantic actions that you can attach to grammar rules that breathe life into your DSEL. I have so much more to say, but I’ll leave the rest for later. In future articles, I’ll describe Proto in more detail, build some simple but useful DSELs and work slowly toward a radical goal[[47]](#footnote-47): the design of a DSEL—to be released shortly—called Boost.Phoenix3, a library that blurs the line between C++ code and data[[48]](#footnote-48), promising to bring the power of [Lisp macros](http://en.wikipedia.org/wiki/Macro_%28computer_science%29#Lisp_macros) to C++ programmers[[49]](#footnote-49). By the time the series is done, you’ll be like [Neo in the Matrix](http://www.imdb.com/video/screenplay/vi1032782617/), seeing your own code as data that you can bend to your will[[50]](#footnote-50).

1. From the Wikipedia entry [Compiler-compiler](http://en.wikipedia.org/wiki/Compiler-compiler): “A typical parser generator associates executable code with each of the rules of the grammar that should be executed when these rules are applied by the parser. These pieces of code are sometimes referred to as **semantic action** routines since they define the semantics of the syntactic structure that is analyzed by the parser. Depending upon the type of parser that should be generated, these routines may construct a parse tree (or AST), or generate executable code directly.” [↩](http://cpp-next.com/archive/2010/08/expressive-c-introduction/1)
2. http://www.oonumerics.org/blitz/benchmarks/ [↩](http://cpp-next.com/archive/2010/08/expressive-c-introduction/2)
3. It is a curious property of strongly-typed expression trees that they have rich structure both in their types and in their values. Any effort to manipulate them means that two parallel computations must occur: one on types and another on values. This is an extra complication that a good DSEL toolkit should handle effortlessly. [↩](http://cpp-next.com/archive/2010/08/expressive-c-introduction/3)

Posted Monday, August 30th, 2010 under [Boost](http://cpp-next.com/archive/category/boost/).

1. lurk : 잠복하다, 도사리다. Hidden in C++ lurks another language := 또 다른 언어가 C++ 의 장막 안에 도사리고 있다. [↑](#footnote-ref-1)
2. cut away : 떼어 내다 [↑](#footnote-ref-2)
3. we do (want) : 우리가 원하는 [↑](#footnote-ref-3)
4. take a close look at ~ : ~ 을 주의 깊게[세심히] 보다. ~ 을 자세히 쳐다보다. [↑](#footnote-ref-4)
5. what they’re good for : 그것들이 무엇에 좋은지 [↑](#footnote-ref-5)
6. a vexing problem : 성가신 문제. [↑](#footnote-ref-6)
7. to precisely nail down : 정확성 있게 명확화 시키기 [↑](#footnote-ref-7)
8. What he came up with : 그가 찾아 내어 놓은 해법 [↑](#footnote-ref-8)
9. it turns up in various guises : 다양한 곳에 나타난다. [↑](#footnote-ref-9)
10. EBNF’s ubiquity is a testament to its simple, elegant power. : 이러한 광범위한 사용은 EBNF의 단순하고 우아한 능력을 보여주는 증거이다. [↑](#footnote-ref-10)
11. just the ticket for anyone ~ but pretty useless for everybody else: 특정 누군가를 위한 입장권일 뿐이고, 다른 이들에게는 전혀 쓸모가 없다. [↑](#footnote-ref-11)
12. part of its appeal : 매력 요소 [↑](#footnote-ref-12)
13. ~ at hand without distracting you with superfluous fluff : 불필요한 군더더기로 산만해 지지 않고 즉시 쓸 수 있도록 [↑](#footnote-ref-13)
14. something that falls *just* outside of EBNF’s purview : EBNF의 범위를 벗어나기*만* 한 것 [↑](#footnote-ref-14)
15. This is the bugaboo of all domain-specific languages : 이게 바로 모든 '영역 특정 언어'가 가지고 있는 무서움 이다. [↑](#footnote-ref-15)
16. ~ so you can escape the confines of your domain when you have to : 다루고 있는 '영역'의 범위를 벗어 나야 할 때는 빠져 나오는 것이 가능하도록 [↑](#footnote-ref-16)
17. Anything that can be done at all (with certain language)- can be done with C++ : (다른 언어로) 할 수 있는 일들은 그 무엇이든 C++ 로 할 수 있다. [↑](#footnote-ref-17)
18. comes with a cost : 비용이 든다 [↑](#footnote-ref-18)
19. many things for which C++ isn’t particularly great out of the box : C++이 돋보이게 특별히 잘한다고 볼 수 없는 많은 것들 [↑](#footnote-ref-19)
20. ~ comes to mind : ~ 이 생각난다. [↑](#footnote-ref-20)
21. C++’s answer is to be great at defining libraries : '라이브러리 작성을 쉽게 하도록 하기'가 C++이 제시하는 답이다. [↑](#footnote-ref-21)
22. The onus then is on library writers to provide ~ : ~을 제공해야 되는 책임은 이제 라이브러리 제작자들의 몫이다. [↑](#footnote-ref-22)
23. fall off a cliff : 낭떠러지에 떨어지다. [↑](#footnote-ref-23)
24. fall back on good old C++ : 전형적인 C++ 로 돌아가다 [↑](#footnote-ref-24)
25. have come up with a creative solution : 창의적인 해결책을 내놨다. [↑](#footnote-ref-25)
26. If you squint a bit : 조금 째려 보면 [↑](#footnote-ref-26)
27. but it captures the flavor : 그 풍미를 담고는 있다. [↑](#footnote-ref-27)
28. -1) see[think] fit (to ~ ) :(~하는 것이) 맞다고 보다; (~하기로) 결정[선택]하다. 28-2) No hard feelings :감정 상하지 말자. [↑](#footnote-ref-28)
29. This is the essence of it: by rigging things so that domain-level concepts can be expressed directly and concisely with C++ expressions, we can make users of our library comfortable and productive :   
     DSEL 의 정수 : '해당 영역의 개념'이 C++표현식 으로 간결하게 직접 표현 될 수 있게 조작을 가해서, 라이브러리 사용자들은 편해질 수 있고 생산성이 향상될 수 있다. [↑](#footnote-ref-29)
30. What’s more, we can often do it while realizing huge wins in performance : 한 술 더 떠서, 우리는 종종 비약적인 성능 향상이 수반되는 DSEL을 만들 수 있다. [↑](#footnote-ref-30)
31. matrix crunching library : 행렬 처리 라이브러리. [↑](#footnote-ref-31)
32. bested FORTRAN : FORTRAN 을 앞지르다. [↑](#footnote-ref-32)
33. you could be forgiven for thinking of Spirit quite simply as a “library : ~Spirit을 아주 단순히 어떤 "라이브러리" 라고 생각하는 것도 무리가 아니다[이해가 된다] [↑](#footnote-ref-33)
34. Imagine what would change if we all really believed to our core that~ : ~ 라는 핵심 내용을 우리 모두가 진정으로 믿는다면 무엇이 바뀔 수 있는지를 상상해 보자. [↑](#footnote-ref-34)
35. a common enough task that tools make the job easier : 툴을 써서 그 작업을 쉽게 할만한 흔한 일. [↑](#footnote-ref-35)
36. An analogy with Spirit is apt : Spirit 을 이용한 유사점 찾기는 적절하다. [↑](#footnote-ref-36)
37. run-of-the-mill : 너무도 평범한 [↑](#footnote-ref-37)
38. It involves a fair bit of ~ : 상당량의 ~를 포함한다. [↑](#footnote-ref-38)
39. a notoriously difficult task : 악명 높게 어려운 일 [↑](#footnote-ref-39)
40. impenetrable and brittle : 해석하기 어렵고 망가지기 쉬운 [↑](#footnote-ref-40)
41. rigorously and robustly : 엄밀하고도 강건하게 [↑](#footnote-ref-41)
42. rip apart expressions, analyze the parts, and reassemble them : 표현식을 분해하고, 분해된 조각들을 분석하고, 분석된 조각들을 재조립한다. [↑](#footnote-ref-42)
43. we cheekily made it look like one[an output expression] : 건방지게 출력 표현식처럼 보이도록 했다. [↑](#footnote-ref-43)
44. What’s To Come : 이후에 다룰 내용들 [↑](#footnote-ref-44)
45. As you might suspect : 믿음이 가지 않겠지만 [↑](#footnote-ref-45)
46. Proto comes with tools for~ : Proto 에는 ~ 를 위한 도구들이 들어 있다. [↑](#footnote-ref-46)
47. work slowly toward a radical goal : 기가 막힌 목표 달성을 위해 서서히 노력하다 [↑](#footnote-ref-47)
48. blurs the line between C++ code and data : C++ 코드와 데이터의 경계를 모호하게 하다 [↑](#footnote-ref-48)
49. promising to bring the power of Lisp macros to C++ programmers : List 메크로의 강력함을 C++ 프로그래머에게 가져다 주겠다는 약속을 하면서 [↑](#footnote-ref-49)
50. seeing your own code as data that you can bend to your will : 마음대로 조작할 수 있는 '데이터' 처럼 '코드'를 바라보면서. [↑](#footnote-ref-50)