# Domain-specific Language Integration with Compile-time Parser Generator Library [*]

Zoltán Porkoláb
Eötvös Loránd University,
Dept. of Programming Languages
and Compilers
H-1117 Pázmány Péter sétány 1/C
Budapest, Hungary
gsd@elte.hu

Ábel Sinkovics
Eötvös Loránd University,
Dept. of Programming Languages
and Compilers
H-1117 Pázmány Péter sétány 1/C
Budapest, Hungary
abel@elte.hu

## Abstract

Smooth integration of domain-specific languages into a general purpose host language requires absorbing of domain code written in arbitrary syntax. The integration should cause minimal syntactical and semantic overhead and introduce minimal dependency on external tools. In this paper we discuss a DSL integration technique for the C++ programming language. The solution is based on compile-time parsing of the DSL code. The parser generator is a C++ template metaprogram reimplementation of a runtime Haskell parser generator library. The full parsing phase is executed when the host program is compiled. The library uses only standard C++ language features, thus our solution is highly portable. As a demonstration of the power of this approach, we present a highly efficient and type-safe version of `printf` and the way it can be constructed using our library. Despite the well known syntactical difficulties of C++ template metaprograms, building embedded languages using our library leads to self-documenting C++ source code.

***Categories and Subject Descriptors*** D.3.2 [*Programming Languages*]: Language Classification – C++; D.3.2 [*Programming Languages*]: Language Classification – Multiparadigm languages

***General Terms*** Languages

***Keywords*** C++ template metaprogram, DSL integration, Haskell, Parser generator

## 1. Introduction

Modern general purpose programming languages have the ability to express regular programming idioms in a fairly convenient way: functions, types, classes and class hierarchies, etc. are used to express the programmer's intention. In most cases these tools are applied when the programmer transfers a solution from a specific problem domain to the general purpose language. Such transformations require not only good programmer skills in the means of the classical programming language terms but a profound understanding of the specific problem domain as well.

As an opposite, *Domain-specific languages* (DSLs) are created to express problems in particular domains only. Using DSLs in specific problem areas have many advantages. DSLs are regularly more expressive in the intended problem domain. The special syntax of a DSL is able to catch errors specific to the problem domain. DSLs often invent new constructs to describe domain problems or they even apply different programming paradigms. Thus, the syntax of a DSL may reflect the usual notations of the domain to make it usable for the domain experts.

As an example, the `SQL` language is good for expressing relational database related problems while general purpose languages lack this clarity. Relational database related errors are easier to detect in queries written in `SQL` syntax because `SQL` follows the declarative paradigm.

Although DSLs are indispensable in their domain, vast majority of the programs execute most of their actions out of that domain. `SQL` might be a perfect solution for describing operations related to relational databases, but database servers will create threads, open network connections, communicate with the operating system in the means of a general purpose programming language. The usual solution is that the desired domain-specific language or languages are used together with a general purpose programming language. Most cases the integration of these languages happens by embedding the DSL(s) into the general purpose language with or without some syntactical quotation.

However, this integration should add minimal syntactical and semantic overhead to the project. Many strategies exist to provide smooth integration of the domain languages and the host language. Some of them applies external frameworks for integration, others are built on language extensions. Only a few solutions are based on standard programming language features like macros or generative language elements.

Not all of these solutions can be applied in an industrial environment. External tools may introduce unwanted dependencies on 3rd party software. Language extensions require translators, precompilers or the modification of the compiler. These are fragile solutions when new language or compiler versions appear. The most portable, manageable solution is based purely on standard language features.

In this paper we introduce a DSL integration technique for the C++ programming language. The solution is based on compile-time parsing of the DSL code. L. Andersson described the construction of a parser generator in Haskell [31]. We followed his approach, but leveraging the well known connection between func-

[*] Supported by TÁMOP-4.2.1/B-09/1/KMR-2010-0003

tional programming languages and C++ template metaprogramming [1, 2] we implemented the parser generator as a C++ template metaprogram library. Thus, the full parsing phase is executed when the host program is compiled. The library uses standard C++ language features only, thus our solution is highly portable.

Defining new DSLs is simple using our solution. Taking advantage of the declarative nature of C++ template metaprogramming the formal syntax of the DSL can be directly expressed in the source code. Thus, the grammatical rules of the DSL are presented explicitly with a minimal syntactical overhead in the source code. The result of this method is a highly self-documenting and maintainable C++ template metaprogram.

As a demonstration of the power of this approach, we discuss a number of practical domain specific languages. As a non-trivial example, we present the construction of a highly efficient and type-safe version of `printf`.

The rest of the paper is organized as follows. In Section 2 we overview C++ template metaprogramming. Current DSL embedding technologies are discussed in Section 3 with their advantages and shortages. We explain our template metaprogram based parser in Section 4 with sufficient implementation details. In Section 5 we evaluate our solution with the help of non-trivial examples. Our paper concludes in Section 6.

## 2. C++ Template Metaprogramming

Templates are key language elements of the C++ programming language [3]. They are essential for capturing commonalities of abstractions without performance penalties at runtime. The most notable example is the Standard Template Library [12] which is now an unavoidable part of professional C++ programs. In 1994 Erwin Unruh wrote a heavily templated program [22] in C++ which didn't compile, however, the error messages emitted by the compiler displayed a list of prime numbers. Unruh used C++ templates and the template instantiation rules to write a program that is "executed" as a side effect of compilation. It turned out that a cleverly designed C++ code is able to utilize the type-system of the language to force the compiler to execute a desired algorithm [25]. These compile-time programs are called C++ *Template Metaprograms* and later have been proved to be a Turing-complete sub language of C++ [5].

C++ template metaprogram actions are defined in the form of template definitions and are "executed" when the compiler instantiates these templates. Their instantiations can instruct the compiler to execute other instantiations, since templates can refer to other templates. This way we get an instantiation chain very similar to a call stack of a runtime program. Recursive instantiations are not only possible but regular in template metaprograms to model loops. Conditional statements, bottoms of recursions, and compile-time decisions are implemented using template specializations [1, 23].

The compile-time decisions can directly affect the compilation itself. A *static assert* is capable of halting the compilation of a program at the point of the error's detection, thus we can avoid an incorrect program to come into being. At the same time, we aspire to create a static assert that contains some sensible error message, thus it is easier for the programmer to find the bug. The simplest way to execute these checks is using a macro defined in [11]. Static asserts are widely used for type checking in C++ templates [2]. Integration of domain-specific languages requires these techniques to detect invalid states in the domain space and to raise custom errors.

Today programmers write metaprograms for various reasons, like implementing *expression templates* [26], where we can replace runtime computations with compile-time activities to enhance runtime performance; *static interface checking*, which increases the ability of the templates to verify that the template parameters meet some requirements, i.e. they form constraints on template parameters [13, 16]; *active libraries* [24], acting dynamically at compile-time, making decisions and optimizations based on programming contexts. Other applications involve embedded domain specific languages as the AraRarat system [8] for typed safe SQL interface and `boost::xpressive` [36] for regular expressions.

## 3. DSL integration techniques

In this section we overview common patterns in technologies currently used for integration of domain-specific languages.

### 3.1 External frameworks

In the following we discuss a few notable solutions for language integration using external frameworks. The common feature of these approaches is that they intent to use some language independent solution. In most cases the source code written in a specific syntax is transformed into a language-neutral internal representation and different DSLs are integrated in this representation. The integrated program can be generated in a desired syntax.

#### 3.1.1 Stratego/XT

The *Stratego/XT* developed in TU Delft is one of the most promising program transformation systems using external toolsets to integrate DSLs. The Stratego/XT metaprogram system [39, 27] contains the Stratego language describing the program transformations and the XT tool set, which executes the transformations and provides a framework for constructing stand-alone program transformation systems.

Source code written in arbitrary syntax can be transformed into *Annotated Term Format* (ATF), an internal representation that bridges the differences between syntactical notations. Steps of transformations are executed on the ATF before a pretty printer generates source code in a required language from it. Parsing and pretty printing is based on an external description, therefore the set of available language syntaxes are extensible. Some languages, like C++ and Java, are already supported.

The Stratego language is based on *strategic term rewriting*. Transformation definitions have two parts: rewriting rules and strategies. *Rewriting rules* describe basic transformation steps. Applications of these rules are controlled using *strategies*. Rewriting rules can be defined in a language independent way, operating on the internal representation. However, this form is often lengthy. A subsystem called *Metaborg* exists to describe the rewriting rules in the source language.

#### 3.1.2 Intentional programming

Current software development often uses high level, domain-specific notations in the design phase, but almost always ends up implementing the program in some programming language. This last step is not only costly and error-prone, but causes recoding the software when some domain-specific content changes. The idea behind *intentional programming* [18, 37] is to separate the domain contents of the software from its implementation in a specific programming language and automatically regenerate the software as its domain contents change.

Intentional programming allows expressing a program using heterogeneous syntax, i.e. the code can appear in the syntax of a general purpose programming language while some of its parts can be expressed in a domain notation when that is more expressive. Lazy evaluation strategies avoid unnecessary parsing-unparsing steps to improve efficiency.

Domain contents can be extended behind classical programming idioms. Comments, version control information or even the full documentation could be integrated into the program and can be displayed on request.

## 3.2 Language extensions

Language extensions are attractive solutions for embedding domain-specific languages. They keep most of the host language syntax and therefore have zero impact on those parts of the code where DSLs are not used. Keywords or even variables from the domain-specific language can be used without any quotation or syntactical marker.

However, there are several problems when more then one domain language is used in the host language: keywords may collide, domain syntax can be ambiguous, etc. Special parsing and context-aware scanning algorithms required in which the scanner uses contextual information to disambiguate lexical syntax [28]. Van Wyk and others have shown the applicability of the extension mechanism.

Language extensions are fragile in many ways. They require either the modification of the compiler or an extensive set of translators or pre-compilers. Although for some languages like Java exists a set of techniques and frameworks to make language extension less painful, other languages, especially C++, are very hard to extend when conformance to the existing language, stability, and efficiency of the generated code are all targeted.

### 3.2.1 Template Haskell

Template Haskell [41] is a non-standard extension to Haskell 98 in GHC. It is a tool for metaprogramming in Haskell, code manipulating abstract syntax trees can be written in a Haskell-like syntax to generate code. The similarity between the template code and standard Haskell code makes it easier for developers to get started with this tool. It is an extension to the standard and is limited to GHC.

## 3.3 New languages designed for extension

Programming languages can be designed for embedding DSLs by allowing the developers to specify custom syntax and semantic and use them in the source code. These languages become host languages for a number of DSLs. Given that this is what these languages are mainly designed for, embedding DSLs into systems written in these languages is straightforward. However, this approach doesn't target the extension of existing and widely used languages. When using them, a new language and development environment has to be adopted by the developers. Another issue with this approach is that even though these languages are designed for DSL embedding, the syntax of the language may still constrain the syntax of the embedded languages to some extent.

### 3.3.1 Converge

Converge [21] is a general purpose programming language designed for embedding DSLs. It is influenced by Python [42] and Icon [43]. Developers can easily extend the language with compile time metaprograms. It doesn't aim for embedding DSLs into existing languages, it is a new language with good DSL integration support.

### 3.3.2 Katahdin

Katahdin [44] is a programming language designed for using different languages in the same code. Developers can define the syntax and semantic of a new language and Katahdin can parse and execute code written in that language.

### 3.3.3 XMF

XMF [45] is a language designed to support Language Oriented Programming, described on the website of XMF. It aims to be a framework for a number of DSLs solving different parts of the problem and tying them together. It's a new language, but uses the Java virtual machine. It can be easily integrated with Java code, however, it's integration with other languages is difficult.

## 3.4 Generative approach

Expression Templates is an advanced technique that C++ library developers use to define embedded mini-languages that target specific problem domains. The technique has been used to create efficient and easy-to-use libraries for linear algebra as well as to define C++ parser generators with readable syntax. But developing such libraries involves writing an inordinate amount of unreadable and unmaintainable template code.

In the following we overview three application examples of expression templates for implementing domain-specific language integration.

### 3.4.1 AraRat

The *AraRat* system targets one of the most important domains: it demonstrates the integration of a relational algebra based language into C++ [8] making the generation of type-safe SQL queries and effective POD types for storing query results possible.

The system works in two phases. In the first phase a little external tool is used to discover the database schema and to generate a set of C++ types and operator overloads to reflect the schema information. In the host language relational expressions are represented as C++ expressions using the overloaded operators. Template metaprogram techniques are used to check the consistency of relational operations and generating result sets in an efficient way.

Its idea is impressive, but the AraRat system has constraints. Its domain is restricted to the domain of relational algebra, mainly for type-safe selections. The domain language has to follow the syntax of C++.

### 3.4.2 Boost::Xpressive

*boost::xpressive* is an advanced, object-oriented regular expression template library for C++ [36]. Regular expressions can be written as strings that are parsed at runtime or as expression templates that are parsed at compile-time. Regular expressions can refer to each other and to themselves recursively, making it possible to build arbitrarily complicated grammars.

Regular expressions are a paragon of domain-specific languages. They are used for a very special purpose, text manipulation, and have a specific, usually implementation-independent syntax. Regular expressions are usually implemented as libraries. Classical regular expression libraries, like `boost::regex`, are powerful and flexible. Patterns are represented as strings which can be specified at runtime. However, it means that syntax errors are not detected until runtime. Additionally, regular expressions are ill-suited for advanced text processing tasks, such as matching balanced, nested tags.

`boost::xpressive` brings these two approaches seamlessly together and occupies a unique niche in the world of C++ text processing. Users can represent regular expressions as strings or can use them as C++ expression templates. In this case regular expressions can be statically bound, hard-coded and syntax-checked by the compiler or dynamically bound and specified at runtime. These regular expressions can refer to each other recursively, matching patterns in strings that ordinary regular expressions cannot.

While `boost::xpressive` behaves similarly to our solution integrating a domain-specific language at compile-time and performing syntax checks on it, its purpose is limited to a pre-defined domain: text manipulation.

### 3.4.3 Boost::Proto

The *boost::proto* library takes one further step forward from `xpressive` in providing a framework for building Domain Specific Embedded Languages in C++ [35]. It provides tools for constructing, type-checking, transforming and executing domain-specific languages expressible as expression templates. Proto provides data structures for representing the expressions and a mechanism for giving additional behaviors and members to them.

Expression trees are built from expressions of the domain-specific languages using operator overloads. Utilities for defining the grammar of the expressions and an extensible set of mechanism for immediate execution and tree transformations are also provided. The use of `boost::proto` for defining the primitives of a domain-specific language radically simplifies the task of integrating DSLs.

The `boost::proto` library is one of the most general existing solutions for embedding domain-specific languages into C++. Unfortunately, it has restrictions. As the expression tree is built up with the help of operator overloads, the domain-specific language has to follow valid C++ expression syntax, i.e. keywords or variables have to be connected to overloaded C++ operators. This is a serious restriction when speaking about general purpose domain languages. In return no quotations should be applied to identify domain language code.

### 3.5 Embeded DSLs

Embedded domain specific languages extend an existing language and make it capable of handling a new problem domain in an efficient way. The right abstraction is different for every problem domain. One language can't capture them all but different embedded languages can provide different abstractions, thus they can provide the right abstraction for every problem domain [7]. Embedded DSLs don't require external tools or changes in the compiler or the runtime environment. When the host language is available on multiple platforms, carefully designed embedded DSLs are portable as well. The embedded language may be limited by the syntax and semantics of the host language.

## 4. Our solution

Our solution is based on the parser introduced in [31]. The paper describes a runtime Haskell parser generator library in detail. We transformed the library to a C++ template metaprogramming library using a structured approach, detailed below. The result is a compile-time parser generator library for C++. In this section we present the details of the translation. The library is available at [46].

### 4.1 Syntax for embedding source codes

The input of the parser is the text to parse, represented as a string. In Haskell a string is a list of characters [14]. In C++ template metaprogramming we cannot handle string literals, we will use a list of characters as well [1]. For example, the string `Hello World!` will be represented in a C++ template metaprogram as:

```
lit_c<char,
  'H','e','l','l','o',' ','W','o','r','l','d','!'>
```

`boost::mpl` has a tool for string definition which simplifies the declaration of compile-time strings [33]:

```
string<'Hell', 'o Wo', 'rld!'>
```

Support for user-defined literals has been proposed to be included in the upcoming C++ standard, C++1x. This proposal contains a solution for the conversion of a string literal to the instantiation of a variadic template [40] function with the characters of the string as template arguments. With the combination of this, `decltype` [40] and the C++ pre-compiler the notation can be simplified to:

```
_S("Hello World!")
```

By using an external translator, such as a trivial Python script, this C++1x behavior can be simulated. An implementation of this script is part of the source tree of our library available at [46]. Using this script we minimize the syntactical overhead while we use only standard C++ language features.

We present how we implemented those features of Haskell which are used by the library. We don't describe every part of the translation here, we focus only on the key elements.

### 4.2 Algebraic types

The Haskell parser described in [31] is based on algebraic data types. Algebraic data types in Haskell have the following form:

```
data <name> [<type arguments>] =
  <constructor name> <constructor arguments> |
  <constructor name> <constructor arguments> |
  ...
```

We implement each constructor by a C++ template. The constructor arguments are the template arguments. For example the constructor `Div Expr Expr` is implemented as

```
template <class Expr1, class Expr2>
struct Div {};
```

We couldn't express Haskell types in C++ template metaprograms, the type of the template arguments is always `class`. Algebraic data types and their arguments have no direct representation in C++ template metaprogramming, only the constructors are implemented.

In Haskell the constructors of algebraic data types act as functions to construct objects. We need to turn their C++ template metaprogramming implementations into functions as well. We can do it by turning them into nullary template metafunctions evaluating to themselves. For example the `Div` function could be enhanced the following way:

```
template <class Expr1, class Expr2>
struct Div
{
  typedef Div<Expr1, Expr2> type;
};
```

This template works with functions expecting a data-type and with functions expecting a nullary template metafunction as well. It behaves as expected in both situations.

As an example for translating algebraic data types we present our translation of Haskell's `Maybe`. In Haskell it's

```
Maybe a = Nothing | Just a
```

In C++ template metaprogramming it's

```
struct Nothing
{
  typedef Nothing type;
};

template <class a>
struct Just
{
  typedef Just<a> type;
};
```

### 4.3 Functions

Haskell builds on currying to represent functions, a function takes exactly one argument. Functions taking multiple arguments are implemented as functions taking 1 argument and returning other functions. For example a function taking 3 arguments is implemented as a function taking 1 argument and returning a function taking another argument and returning a function taking a third argument returning the value of the 3 argument function.

In our C++ template metaprogramming representation of the Haskell functions we didn't represent currying: we implemented Haskell functions as functions taking multiple arguments. Haskell functions have the form of

```
f :: <arg 1> -> ... -> <arg n> -> <result type>
```

which we implemented in C++ template metaprogramming with template metafunctions or template metafunction classes depending on how we wanted to use them:

```
template <class arg1, class arg2, ..., class argn>
struct f
  // ...
{};
struct f      // alternative solution
{
  template <class arg1, ..., class argn>
  struct apply
    // ...
  {};
};
```

The result of the function is the value of the template metafunction or metafunction class. Functions are first-class citizens in Haskell, they can be passed around as data values. In C++ template metaprogramming we can do the same with template metafunction classes. Thus, functions in the library that were arguments or values of other functions we implemented as template metafunction classes, not as simple template metafunctions. `boost::mpl` provides tools which can transform template metafunctions into template metafunction classes in cases we need to turn a template metafunction into a first-class citizen.

## 4.4 Parsers

Parsers are functions with the following signature:

```
type Parser a = String -> Maybe (a, String)
```

A parser takes the input string as its argument and returns a parsed object and the remaining part of the input when it accepts a prefix of the input string. It returns `Nothing` when it rejects the input string. Note that the second element of the tuple is always a postfix of the input string.

A tuple with two elements can be implemented by a pair of classes. A pair data structure exists in `boost::mpl` which we can use. A parser is a function in the Haskell library, so it's a template metafunction in C++ template metaprogramming. Here is the definition of one of the basic parsers in Haskell:

```
char :: Parser Char
char (c:cs) = Just (c, cs)
char [] = Nothing
```

and in C++ template metaprogramming:

```
struct one_char
{
  template <class s>
  apply :
    eval_if<
      typename empty<s>::type,
      Nothing,
      Just<build_pair<front<s>, pop_front<s> > >
    >
  {};
};
```

Note that in C++ we had to call it `one_char` because `char` is a reserved word. `build_pair` is a helper metafunction taking nullary metafunctions as arguments and building a pair structure from them. We had to use `eval_if` instead of pattern matching. Even though C++ templates have excellent pattern matching support [1] when we're constructing code from the building blocks `boost::mpl` provides, we can't use it. To be able to pass `one_char` to parser combinators, which are template metafunctions, we had to implement it as a template metafunction class.

Some parsers have arguments. The Haskell library builds on currying in Haskell: parsers taking arguments are functions with multiple arguments and the input string is always the last argument. By applying all arguments except the input string to these functions we get a parser: a function taking an input string as an argument and parsing it. For example `return` is a parser with an argument:

```
return :: a -> Parser a
return a cs = Just(a, cs)
```

Its C++ template metaprogramming implementation has to be a metafunction returning a parser, which is a metafunction:

```
template <class a>
struct return_
{
  struct type
  {
    template <class cs>
    struct apply : Just<pair<a, cs> > {};
  };
};
```

## 4.5 Parser combinators

Complex parsers are built by combining basic parsers. The Haskell library uses parser combinators which are parsers taking other parsers as arguments. For example the Haskell library defines a ? operator which is an infix operator: its left argument is a parser, its right argument is a predicate providing a boolean value for each result of the parser. We implemented it with a metafunction taking two metafunction classes, a parser and a predicate, as arguments and returning a parser:

```
template <class m, class p>
struct accept_when
{
  // This metafunction class is the value
  // of the accept_when metafunction
  struct type
  {
    template <class cs>
    struct apply :
      lazy_eval_if<
        equal_to<
          typename apply<m, cs>::type,
          Nothing
        >,
        nothing,
        lazy_eval_if<
          apply<p, just_value<apply<m, cs> > >,
          apply<m, cs>,
          nothing
        >
      >
    {};
  };
};
```

Note that the application of an argument to a function in Haskell, which is writing the function and the operand after each other, can be implemented using the `apply` metafunction in template metaprogramming.

This function can be used the same way it's used in the Haskell library. For example we can use it to implement the `digit` function:

```
template <class cs>
struct digit :
  accept_when<one_char, isDigit>::type {};
```

`isDigit`'s C++ template metaprogramming implementation is straightforward but lengthy, we're not going to present it here.

### 4.6 Recursive functions

Recursive functions can be translated as well, template metafunctions can call themselves. We present our implementation of `iter` here as an example, other recursive functions can be translated similarly. The Haskell implementation of it is

```
iter :: Parser a -> Parser [a]
iter m = m # iter m >-> cons ! return []
```

while our translated implementation is

```
struct iter
{
  template <class m>
  struct apply :
    parser::one_of< // !
      parser::transform< // >->
        parser::sequence< // #
          m,
          boost::mpl::apply<parser::iter, m>
        >,
        parser::cons
      >,
      parser::return_<boost::mpl::list<> >
    >
  {};
};
```

Note that we combined the C++ template metaprogramming implementations of the operators the Haskell implementation uses the same way the Haskell code does it. In the example above we added the original names of the operators as comments to the functions.

The whole Haskell library can be translated to C++ template metaprograms following this approach, we don't present every step here. As a result we get the same functionality at compile-time in C++ the Haskell library provides at runtime.

## 5. Evaluation

Embedded languages can be compiled as part of the C++ compilation process using template metaprograms and used by the host program. We have built a library for constructing these compile-time parsers. To demonstrate the power of the library we present a non-trivial DSL and its compile-time parser using our library.

### 5.1 Type-safe printf

Though the `printf` function of the standard C library is efficient and easy to use, it's not type-safe, hence mistakes of the programmer cause undefined behavior at runtime. Some compilers, such as gcc, type check `printf` calls and emit warnings in case there is a type mismatch between the format string and the parameter types or the number of parameters are different than specified but this method is not widely available. To overcome these problems C++ introduced *streams* as a replacement of `printf`, which are type-safe but have significant runtime and some syntactical overhead.

In most cases the format string of `printf` is a static string literal in which case its value is available at compile-time, thus the compiler could do type-checking and spot misuses of the function. `boost::mpl` [33] supports compile-time strings which could be used to represent the format string.

Stroustrup presents [40] a type-safe variant of `printf` using variadic template functions which are part of the upcoming standard, C++1x [20]. That implementation, originally presented in [9], uses runtime format strings and transforms `printf` calls to write to C++ streams at runtime. See the example:

```
printf("Hello %s!", "John");
```

The method presented in [9] does the following at runtime:

```
std::cout
  << 'H' << 'e' << 'l' << 'l'
  << 'o' << ' ' << "John" << '!';
```

His solution was primarily written to demonstrate the use of variadic templates, that is why printing the format string is done character by character, making the process extremely slow. This method can be optimized in the following, more efficient way:

```
std::cout << "Hello " << "John" << "!";
```

Our solution is a type-safe version of `printf` that avoids the runtime overhead of streams by using the `printf` function of the C library, but with the guarantee of type-safety. Type-safe `printf` calls can be written the following way:

```
safe::printf<_S("Hello %s!")>("John");
```

`Hello %s!` is the format string and `John` is the argument for `%s`. `safe::printf` has all the information needed to verify the correctness of the type of the argument or arguments at compilation time. At runtime it calls

```
printf("Hello %s!", "John");
```

The compile-time verification guarantees that there will be no problems with the number or type of the arguments. When the number or type of the arguments are incorrect and would lead to runtime errors, `safe::printf` emits compilation errors. The template metafunction verifying the arguments has no runtime, only compile-time overhead. The body of `safe::printf` consists of a call to `printf`, which is likely to be in-lined, thus using `safe::printf` has no runtime overhead compared to `printf`, it has the same runtime performance.

We have measured the speed of these operations and of the normal `printf` used by our implementation. We printed the following text and its `std::cout` equivalents:

```
printf("Test %d stuff\n", i);
```

The text was printed 100 000 times and the speed using the `time` command on a Linux console was measured. The result of the measurement is presented in Table 1. The `printf` function, which is used by the type-safe implementation, is almost four times faster than the example at [40] and more than two times faster than the optimized version of the example.

The grammar of the format string is a complex domain specific language and the validator metafunction has to parse it, thus implementing a type-safe `printf` is difficult without a compile-time parser.

| Method used | Time |
|---|---|
| std::cout for each character | 0,573 s |
| normal std::cout | 0,321 s |
| printf | 0,152 s |

**Table 1.** Elapsed time

## 5.2 Building the parser for the type-safe printf

The library is available on-line at [46]. The `printf` function is in the `mpllibs::printf` namespace, but to simplify it's usage we assume that the following alias is defined:

```
namespace safe = mpllibs::printf;
```

First we need to write the `safe::printf` template functions users of `safe::printf` can call. These template functions take the format string and the types of the arguments as template arguments, the arguments themselves as runtime arguments. They build a list describing the expected argument types based on the format string and verify the types of the arguments `safe::printf` was called with using this list. We can use BOOST_STATIC_ASSERT [11] to break the compilation in case they don't match. Since variadic templates will be introduced only in C++1x, and they are not widely available yet, we have to create different template functions for different numbers of arguments. Here are a few examples:

```
template <class formatString, class T1>
int safePrintf(T1 t1_) {
  BOOST_STATIC_ASSERT((ValidPrintf<
    formatString, boost::mpl::list<T1>
  >::value));
  printf(
    boost::mpl::c_str<
      formatString>::type::value, t1_);
}
template <class formatString,class T1,class T2>
int safePrintf(T1 t1_, T2 t2_) {
  BOOST_STATIC_ASSERT((ValidPrintf<
    formatString, boost::mpl::list<T1, T2>
  >::value));
  printf(
    boost::mpl::c_str<
      formatString>::type::value, t1_, t2_);
}
// ...
```

These functions can be automatically generated by the boost preprocessor library [34]. `ValidPrintf` is a template metafunction taking a format string and a list of argument types as arguments and evaluating to a boolean value describing the correctness of the argument list based on the format string. It generates a list describing the expected argument types by parsing the format string and verifies the list of actual types. The function evaluates to `true` when the two lists match and `false` when they don't. It can be implemented the following way:

```
template <class F, class ArgTypes>
struct ValidPrintf :
  MatchLists<
    typename
      BuildList< PrintfParser::apply<F> >::type,
    boost::mpl::identity<ArgTypes>
  > {};
```

`PrintfParser` is a metafunction implementing the parser that builds the list of expected types by parsing the format string, `MatchLists` implements the list matching. The expected type of a parameter can be described by the following triple:

- A boolean value telling if there will be an extra preceding integer parameter describing the display length.
- A boolean value telling if there will be an extra preceding integer parameter describing the precision.
- A parameter describing the type of the argument.

The expected type of a parameter's value has to be described using placeholders, given that multiple types can be accepted in many places. We can create placeholders the following way:

```
struct ExpectCharacter {};
struct ExpectString {};
struct ExpectDouble {};
struct ExpectUnsignedInteger {};
// ...
```

One of these placeholders can be the third element of the triple. `PrintfParser` parses the format string and builds the list of these triples. `BuildList` takes the list of triples and transforms it into a list of expected types by replacing each tuple with a number of expected types: an `ExpectUnsignedInteger` element for each `true` value as the first and second elements of the triple followed by the third element of the triple, the type of the argument itself. We don't present the implementation of this metafunction here due to the lack of space. `MatchLists` matches this list against the list of the types of the parameters `safe::printf` was called with. We don't present the implementation of this function here either. The full implementation can be downloaded from [46]. We focus on the implementation of `PrintfParser` to demonstrate how it can be built using our parser generator library.

To build `PrintfParser` we have to define a grammar for `printf` using our parser generator library. Our library provides a `build_parser` metafunction, which builds the parser from this grammar:

```
typedef build_parser<S> PrintfParser;
```

`S` is the start symbol of the grammar. We show how this grammar can be built using our library. The grammar we use is based on [38].

```
S ::= CHARS (PARAM CHARS)*
PARAM ::= '%' FLAG* PRECISION FORMAT
FORMAT ::= 'h' FORMAT_HFLAG | 'l' FORMAT_LFLAG
           | 'L' FORMAT_LLFLAG | FORMAT_NO_FLAG
FORMAT_LLFLAG ::= 'e' | 'E' | 'f' | 'g' | 'G'
FORMAT_LFLAG ::= 'c' | 'd' | 'i' | 'o' | 's'
                | 'u' | 'x' | 'X'
FORMAT_HFLAG ::= 'd' | 'i' | 'o' | 'u' | 'x' | 'X'
FORMAT_NO_FLAG ::= 'c' | 'd' | 'i' | 'e' | 'E'
                 | 'f' | 'g' | 'G' | 'o'
                 | 's' | 'u' | 'x' | 'X'
                 | 'p' | 'n' | '%'
PRECISION ::= '.' WIDTH | NONE
WIDTH ::= INTEGER | '*' | NONE
INTEGER ::= DIGIT+
DIGIT ::= '0' | '1' | '2' | '3' | '4'
        | '5' | '6' | '7' | '8' | '9'
FLAG ::= '-' | '+' | ' ' | '#' | '0'
CHARS ::= (not '%')*
NONE ::= epsilon
```

`CHARS` represents non-interpreted characters, `PARAM` represents one parameter to be substituted. Our parser has to skip all non-interpreted characters, determine the type required by the `PARAM` parts and build the list of these types.

The parser for CHARS can be built the following way:

```
struct Chars :
  any<second_of<except<lit_c<'%'>, int>,
  one_char> > {};
```

The except parser ensures that parsing stops at the first % character, one_char parses one character. second_of throws away the result of the except parser and keeps only the result of one_char, which is the parsed character. any repeats this parser as long as it can, until the next % character or the end of the string is reached. The result of this parsing will be the parsed string itself.

Note the similarities between the definition of CHARS in the grammar and it's implementation: except implements the not part of the rule, lit_c<'%'> implements '%' and any implements the Kleene-star. The rest of the elements of Chars tells the library what the result of parsing should be. We'll present the implementation of other rules from the grammar as well, we won't mention these mappings for the rest of them.

We're done with parsing non-interpreted characters, we build a parser for PARAM as well. The result of parsing a PARAM element will be a triple described above. Using these two parsers we construct the start symbol of the grammar:

```
struct S :
  keep_second<Chars,
    any< keep_first<Parameter, Chars> >
  > {};
```

This throws away the result of parsing non-interpreted characters and keeps only the result of parsing the parameters. any constructs a list of the results, thus it builds the list of triples we need as the result of parsing a format string. Parameter parses one parameter, it implements PARAM. It can be implemented the following way:

```
struct Parameter :
  keep_second<
    lit_c<'%'>,
    keep_second<
      any<Flag>,
      sequence< Width,
      sequence<Precision, Format>
    > > > {};
```

lit_c parses one % character, Flag parses a FLAG element, Width parses a WIDTH element, Precision parses a PRECISION element, Format parses a FORMAT element. Flags don't affect the type of the parameter, we can safely throw the result of parsing them away. Triples will be pairs of pairs, built by sequences. The result of Width and Precision will be boolean values, the first two elements of the triples, the result of Format will be a placeholder for the expected type.

Width can be implemented the following way:

```
struct Integer : any1<digit> {};
struct Width :
  one_of<
    always<Integer, mpl::false_>,
    always<lit_c<'*'>, mpl::true_>,
    return_<mpl::false_>
  > {};
```

Integer is a non-empty list of digits. always takes a parser as an argument and when the parsers succeeds, it throws the result of the parsing away and replaces it with the second argument, true_ or false_ in this case. return_ doesn't parse anything, but always succeeds and the result of the parsing is the argument of return_.

The result of Width is true_ when it parses a * character and false_ otherwise.

The implementation of Precision reuses Width:

```
struct Precision :
  one_of<
    second_of<lit_c<'.'>, Width>,
    return_<mpl::false_> > {};
```

All we have left is the implementation of Format. It has to accept an optional flag and a character specifying the type of the parameter. The set of acceptable parameter types varies based on the flag. We need to parse the flag first and the parameter type afterwards.

```
struct Format :
  one_of<
    keep_second<lit_c<'h'>, Format_hFlag>,
    keep_second<lit_c<'l'>, Format_lFlag>,
    keep_second<lit_c<'L'>, Format_LFlag>,
    Format_NoFlag > {};
```

The parsers beginning with Format_ handle the different set of acceptable parameter types based on the flag. Format parses the flag first and uses the appropriate Format_ parser. The implementation of the Format_ functions is simple, we present one of them here, the rest of them can be implemented in a similar way.

```
struct Format_LFlag :
  one_of<
    always<lit_c<'e'>, ExpectLongDouble>,
    always<lit_c<'E'>, ExpectLongDouble>,
    always<lit_c<'f'>, ExpectLongDouble>,
    always<lit_c<'g'>, ExpectLongDouble>,
    always<lit_c<'G'>, ExpectLongDouble>,
    return_<RejectAll> > {};
```

We use a special placeholder for expected types, RejectAll, when none of the acceptable format specifiers can be found. This placeholder will not match against any parameter type, thus the matching algorithm will commit a compilation error and reject the malformed format string.

The implementation of a type-safe printf is now complete, it can deal with the entire syntax of printf format strings. We could implement it following the grammar of printf. The non-terminal elements of the grammar are represented by template classes. Sequence, selection and repetition constructs used in the grammar are implemented by metafunctions provided by our parser generator library, thus the grammar itself can be easily reconstructed from this implementation. Using our parser generator library we could easily implement a non-trivial grammar and we got a C++ template metaprogram with self-documenting source code.

### 5.3 Measuring the type-safe printf

We measured the compilation and execution time of the type-safe printf and compared it to streams, the type-safe formatting solution of the standard library. We did the following measurements:

- Formatting 1, 2, 3, 4 and 5 floating point numbers using one printf call
- Formatting 1, 2, 3, 4 and 5 different argument types, such as integer and floating point numbers, strings, etc. We were using one printf call to format all of them.
- In real code there will be multiple printf calls. We measured what happens when we have multiple printf calls in one compilation unit by repeating the previous measurement, but using multiple printf calls: instead of measuring with *n* different argument types, we made printf calls with *1*, *2*, ..., *n* arguments one after each other.

We run the measurements on a Linux command line. The machine we did the measurements on had an 1.6 GHz Atom processor and 1 GB memory. We were using gcc 4.4.3 and we didn't use any optimization. To measure the execution time, we repeated the formatting 100 000 times in a `for` loop. This loop didn't affect compilation time. We measured the execution time of using the original `printf` to see if `safe::printf` has any overhead.

| Number of arguments | Homogeneous arguments | Heterogeneous arguments | Multiple calls |
|---|---|---|---|
| 0 | 3.27 | 3.27 | 3.27 |
| 1 | 8.93 | 8.93 | 8.93 |
| 2 | 9.58 | 9.93 | 10.73 |
| 3 | 10.30 | 11.06 | 15.12 |
| 4 | 11.36 | 12.60 | 24.33 |
| 5 | 12.65 | 14.27 | 40.79 |

**Table 2.** Compilation time of safe::printf in seconds

| Nr. of args. | stream | safe::printf | printf |
|---|---|---|---|
| 0 | 0.00 | 0.00 | 0.00 |
| 1 | 0.32 | 0.03 | 0.03 |
| 2 | 0.57 | 0.05 | 0.05 |
| 3 | 0.85 | 0.08 | 0.08 |
| 4 | 1.45 | 0.08 | 0.08 |
| 5 | 1.44 | 0.08 | 0.08 |

**Table 3.** Execution time with homogeneous arguments in seconds

| Nr. of args. | stream | safe::printf | printf |
|---|---|---|---|
| 0 | 0.00 | 0.00 | 0.00 |
| 1 | 0.32 | 0.03 | 0.03 |
| 2 | 0.34 | 0.03 | 0.03 |
| 3 | 0.34 | 0.03 | 0.03 |
| 4 | 0.35 | 0.03 | 0.03 |
| 5 | 0.41 | 0.04 | 0.03 |

**Table 4.** Execution time with heterogeneous arguments in seconds

| Nr. of args. | stream | safe::printf | printf |
|---|---|---|---|
| 0 | 0.00 | 0.00 | 0.00 |
| 1 | 0.32 | 0.03 | 0.03 |
| 1..2 | 0.84 | 0.06 | 0.06 |
| 1..3 | 1.60 | 0.09 | 0.09 |
| 1..4 | 2.72 | 0.09 | 0.09 |
| 1..5 | 4.22 | 0.09 | 0.09 |

**Table 5.** Execution time with heterogeneous arguments and multiple calls

As we can see in Table 2, type checking has a cost at compile-time but the measurement of the execution times in Tables 3, 4 and 5 shows that it runs significantly faster than streams with the same type-safety guarantees. We have also seen that our type-safe `printf` runs as fast as the original one, it has no run-time overhead.

### 5.4 Alternation at compile-time

The type constructed as the result of parsing depends on the embedded code. We can easily construct a parser that takes a number as its input and returns the `int` or `double` type, depending on which type of variable could store that specific number. Here is the parser:

```
typedef
  parser::keep_second<
    parser::any1<parser::digit>,
    parser::if_<
      parser::sequence<
        parser::lit_c<'.'>,
        parser::any<parser::digit>
      >,
      double,
      int
    >
  > S;
typedef parser::build_parser<S> Num;
```

And here is how it can be used:

```
Num::apply<_S("13")>::type // int
Num::apply<_S("11.13")>::type // double
```

In the example above we made compile-time decisions based on the parsing result of the embedded language. This scenario shows the ability of host code adaption depending on the domain-specific code.

## 6. Conclusion

Smooth integration of domain-specific languages into a general purpose programming language is not an easy task. A domain specific language is intended to express the domain knowledge in the best possible way, thus its syntax may radically differ from the host language. A general case of language integration therefore could be solved only by applying a parser infrastructure. External tools and frameworks exist for the problem but they introduce unwanted dependency on third party tools. The best self-containing solution should use only standard language features and should use only a minimal set of external tools, if any other than the compiler of the host language.

Our solution full-fills most of these requirements. We created a C++ template metaprogram library with the meaningful translation of a similar Haskell run-time tool, which implements a full-featured parser infrastructure. Domain-specific language code is presented for the parser as template arguments and evaluated during the compilation of the host code. The result of the parsing process is a set of C++ classes which could be used for further compile-time decisions in the template metaprogramming environment. We presented a number of examples to show the usability of our library.

The library uses only standard C++ language features, thus our solution is highly portable. It has a minimal syntactical overhead which can be eliminated by a trivial transformation on the source code. This transformation later could be avoided as the next C++ standard will introduce user-defined custom literals supporting the straightforward presentation of the embedded domain-specific language syntax.

A large number of languages can be embedded into C++ source code by using compile-time parsers. The embedded source code can be a compile-time string parsed by a metaprogram as part of the compilation process. We are currently implementing a compile time parser for embedded SQL which can validates the SQL queries and builds the corresponding C++ classes. This way we are planning to create "safe" SQL queries where query strings are guaranteed to be valid SQL queries. This solution can provide safety against SQL injection as well.

We are also working on the full automation of the transformation process from Haskell to C++ template metaprograms. Thus, the C++ template metaprogram community could leverage from the large set of Haskell run-time code base at compile-time.

## References

[1] D. Abrahams, A. Gurtovoy, C++ template metaprogramming, Concepts, Tools, and Techniques from Boost and Beyond, Addison-Wesley, Boston, 2004.

[2] A. Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied, Addison-Wesley, 2001.

[3] ANSI/ISO C++ Committee, Programming Languages – C++, ISO/IEC 14882:1998(E), American National Standards Institute, 1998.

[4] K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, T. Veldhuizen, Generative Programming and Active Libraries, Springer-Verlag, 2000.

[5] K. Czarnecki, U. W. Eisenecker, Generative Programming: Methods, Tools and Applications, Addison-Wesley, 2000.

[6] G. Dos Reis, B. Stroustrup, Specifying C++ concepts, Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2006, pp. 295-308.

[7] Paul Hudak, Building domain-specific embedded languages, ACM Computing Surveys (CSUR), Volume 28, Issue 4es (1996)

[8] Y. Gil, K. Lenz, Simple and Safe SQL queries with C++ templates, In: Charles Consela and Julia L. Lawall (eds), Generative Programming and Component Engineering, 6th International Conference, GPCE 2007, Salzburg, Austria, October 1-3, 2007, pp.13-24.

[9] Douglas Gregor, Jaakko Järvi, Variadic templates for C++, Symposium on Applied Computing, Proceedings of the 2007 ACM symposium on Applied computing, Seoul, Korea (2007), pp.1101-1108.

[10] D. Gregor, J. Järvi, J.G. Siek, G. Dos Reis, B. Stroustrup, A. Lumsdaine, Concepts: Linguistic Support for Generic Programming in C++, In Proceedings of the 2006 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'06), October 2006.

[11] B. Karlsson, Beyond the C++ Standard Library, An Introduction to Boost, Addison-Wesley, 2005.

[12] D. R. Musser, A. A. Stepanov, Algorithm-oriented Generic Libraries, Software-practice and experience 27(7), 1994, pp.623–642.

[13] B. McNamara, Y. Smaragdakis: Static interfaces in C++. In First Workshop on C++ Template Metaprogramming, October 2000

[14] B. O'Sullivan, J. Goerzen, D. Stewart, Real World Haskell, O'Reilly, 2008. ISBN: 978-0-596-51498-3

[15] Z. Porkoláb, J. Mihalicza, Á. Sipos, Debugging C++ template metaprograms, In: Stan Jarzabek, Douglas C. Schmidt, Todd L. Veldhuizen (Eds.): Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings. ACM 2006 ISBN 1-59593-237-2, pp. 255–264.

[16] J. Siek and A. Lumsdaine: Concept checking: Binding parametric polymorphism in C++, In First Workshop on C++ Template Metaprogramming, October 2000

[17] J. Siek, A. Lumsdaine, Essential Language Support for Generic Programming, Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation, New York, USA, pp 73–84.

[18] C. Simonyi, M. Christerson, S. Clifford, Intentional software, In Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, October 22-26, 2006, Portland, Oregon, USA, pp. 451–465.

[19] B. Stroustrup, The C++ Programming Language Special Edition, Addison-Wesley, 2000.

[20] B. Stroustrup, Evolving a language in and for the real world: C++ 1991-2006. ACM HOPL-III. June 2007

[21] Laurence Tratt, The Converge programming language, Technical report TR-05-01, Department of Computer Science, King's College London, 2005.

[22] E. Unruh, Prime number computation, ANSI X3J16-94-0075/ISO WG21-462.

[23] D. Vandevoorde, N. M. Josuttis, C++ Templates: The Complete Guide, Addison-Wesley, 2003.

[24] T. Veldhuizen, D. Gannon, Active libraries: Rethinking the roles of compilers and libraries. In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientic and Engineering Computing (OO'98). SIAM Press, 1998 pp. 21–23

[25] T. Veldhuizen, Using C++ Template Metaprograms, C++ Report vol. 7, no. 4, 1995, pp. 36-43.

[26] T. Veldhuizen, Expression Templates, C++ Report vol. 7, no. 5, 1995, pp. 26-31.

[27] E. Visser, Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9. In C. Lengauer et al., editors, Domain-Specific Program Generation, vol. 3016 of Lecture Notes in Computer Science, pp. 216–238. Spinger-Verlag, June 2004.

[28] E.R. Van Wyk, A.C. Schwerdfeger, Context-aware scanning for parsing extensible languages, Proceedings of the 6th international conference on Generative programming and component engineering, October 01-03, 2007, Salzburg, Austria, pp. 63-72.

[29] M. Zalewski, A. P. Priesnitz, C. Ionescu, N. Botta, and S. Schupp, Multi-language library development: From Haskell type classes to C++ concepts, In MPOOL 2007 Ecoop workshp, 2007.

[30] I. Zólyomi, Z. Porkoláb, Towards a template introspection library, LNCS Vol.3286 (2004), pp.266-282.

[31] L. Andersson: Parsing with Haskell, October 28, 2001, `http://www.cs.lth.se/eda120/assignment4/parser.pdf`

[32] The boost lambda library. `http://www.boost.org/doc/libs/1_39_0/doc/html/lambda.html`

[33] The boost metaprogram libraries. `http://www.boost.org/doc/libs/1_39_0/libs/mpl/doc`

[34] The boost preprocessor metaprogramming library. `http://www.boost.org/doc/libs/1_41_0/libs/preprocessor/doc/index.html`

[35] The boost proto library. `http://www.boost.org/doc/libs/1_37_0/doc/html/proto.html`

[36] The boost xpressive regular library. `http://www.boost.org/doc/libs/1_38_0/doc/html/xpressive.html`

[37] The Intentional Software. `http://intentsoft.com/`

[38] The printf grammar. `http://www.cplusplus.com/reference/clibrary/cstdio/printf/`

[39] The Stratego Program Transformation Language. `http://strategoxt.org/`

[40] Bjane Stroustrup's C++0x FAQ, `http://www.research.att.com/~bs/C++0xFAQ.html`

[41] Template Haskell `http://www.haskell.org/haskellwiki/Template_Haskell`

[42] Python Programming Language `http://www.python.org`

[43] The Icon Programming Language `http://www.cs.arizona.edu/icon`

[44] Katahdin `http://www.chrisseaton.com/katahdin`

[45] The XMF programming language `http://itcentre.tvu.ac.uk/ clark/xmf.html`

[46] The source code of mpllibs `http://github.com/sabel83/mpllibs`