Embeddable Framework for Syntax-Safe Source Code Generation

Evgenii Kotelnikov
Department of Programming Technology
Faculty of Applied Mathematics & Control Processes
St. Petersburg State University
evgenii.kotelnikov@student.spbu.ru

ABSTRACT
Source code generation is an essential part of wide range of applications, including transcompilers, parser generators and preprocessors. The common technology based on code templates is often cumbersome and error-prone. This paper introduces syngen1, an automatic tool, written in Scheme, designed to produce frameworks for building syntactically correct fragments of source code.

Categories and Subject Descriptors
D.2.3 [Software Engineering]: Coding Tools and Techniques; D.3.4 [Software Engineering]: Processors—source code generation

General Terms
Theory, Languages

Keywords
Source code generation, context-free grammar, algebraic data types, Scheme

1. INTRODUCTION
Generative programming is a well-studied technique of reducing complexity of programming systems and improving programmer’s productivity [1]. Its main concept is to synthesize low-level programs from high-level domain-specific definitions. The nature of this synthesis is tightly bounded with the nature of the programming system environment. Possible ways of implementation include transformation systems, extensible programming systems, metaprogramming techniques and others [2].

Probably the most widely used approach is direct procedure of solution space program source code. Here and further, it will be referred as source code generation.

String.format('SELECT * FROM "%s" WHERE first_name = "%s" AND last_name = "%s",
            tableName, firstName, lastName);

Some sources refer to the same problem as pretty-printing, however different term is used in this work to emphasize, that it is not limited to concerns of code indentation.

Source code generation naturally appears as the final stage of workflow in the following classes of applications.

1. Transcompilers (also referred as source-to-source compilers) translate code in one textual programming language to another.
2. Preprocessors and pretty-printers refine provided source code in order to make it more readable.
3. Parser generators produce implementation of parsing algorithms that recognize specified grammar.
4. Refactoring tools automatically migrate legacy code to new underlying programming language or an API that breaks backward compatibility.

Nevertheless, the scope of application of source code generation techniques is not limited to specialized single-purpose utilities. In fact, wide range of everyday software exploits small version of the same idea. Network applications, that compose high-level network protocol messages actually generate fragments of code according to some data format definition, and typical database applications produce SQL code.

Source code generation is usually performed either by employing code templates or by folding hierarchical structures, similar to syntactic trees of code fragments. The former approach contains essential drawbacks that are in focus of this paper.

Consider an example of database application, that performs SQL queries. The template-based approach of generating SQL code exploits either template engine utilities or just string formatting functions.

Although this approach may work well for a small application with flat logic, it becomes nearly unmaintainable in more complicated cases.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2012 ACM 978-1-4503-1191-5 ...$10.00.
Firstly, since syntactic correctness of produced code is left to a programmer, it could become really hard to provide it when the code is composed in different parts of the program. Various techniques of static string analysis were developed [3, 13] to assist in some cases, but in general, problem remains undecidable.

Secondly, template-based approach forces a programmer to mix code generation logic with syntactic issues of producing language, that leads to tangling templates and increasing the odds of making an error.

Finally, it becomes rather hard to provide consistent code style, i.e. unified rules for linebreaks, whitespaces and indentation.

In contrary, tree-folding approaches isolate concerns of concrete syntax representation and allow to manage correctness of output code by construction of values, representing code fragments.

```java
select(tableName, and(eq("first_name", firstName), eq("last_name", lastName)));
```

A set of definitions of those structures forms a source code generation framework.

In contrast of code templates, this approach does not provide any common solution for building source of an arbitrary language, because the structure of the framework is directly connected to syntactic structure of the language. At the same time, development of such a framework is both tedious and error-prone. This paper is an attempt to automate this process.

2. RELATED WORK

Both theoretical investigations and real-world code generators implementations are available.

M. van der Brand and E. Visser presented [14] a formatter generator, designed mainly for the purpose of automatic code formatting and documentation. It converts context-free grammar to ASF+SDF Metaenvironment specification, that can further be translated into a standalone C program.

DGL [9] is a language for generating test data by probabilistic context-free grammar.


JavaGen [8] provides an API that allows the application to create a formal model of a Java program by instantiating a metamodel of the Java language.

qretty [4] is a library, that builds object-oriented API for generating source code by annotated grammar. Grammar is defined inside Ruby source code, and Ruby is the only supported target programming language.

3. OBJECTIVE

Although there are a few code generators for particular programming languages referred above, no common solution was introduced. The objective of this work is to provide general-purpose tool for producing source code generation framework for any given formal language and unrestricted set of target languages. A key requirement for the framework is to always produce syntactically correct fragments of code. For that reason formal language is given by its grammar. It seems natural to use context-free grammar as the most prevalent in real-world languages.

To provide framework implementation in arbitrary target programming language, translator design is given as formal model of input grammar transformation.

Source code generation framework normally consists of definitions of data structures, representing code structural elements, along with a printer function, that folds code fragment to their concrete syntax. The structure of code is often expressed by abstract syntax trees (AST). However, neither is it clear how to extract AST from language grammar, nor how to ensure syntax-safe folding of code fragment. In the present case, concrete syntax tree (or "parsing tree") fits better as it's possible to directly map parsing tree of arbitrary code to some hierarchical data structures.

To manage correct composition of values of these data structures it is proposed to establish one-to-one correspondence between non-terminals and types in some type system, and then map relations between non-terminals to relations between types. The next section discloses characteristics of a type system capable of expressing such relations and the mapping itself.

4. RATIONALE

Context-free grammar is defined [6] as a tuple \((V, T, P, S)\), where \(V\) is a set of non-terminal symbols \(V\), \(T\) the terminals \(\alpha\), \(P\) the set of production rules of form \(V_i \rightarrow \omega_1 \ldots \omega_n\) and \(S\) the start symbol.

There are four kinds of relations occurring between non-terminals.

1. Non-terminal production consists solely of terminals, so non-terminal doesn't depend on values of other non-terminals.
2. Non-terminal has one production and depend on values of all non-terminals appearing in it.
3. Non-terminal has multiple productions and depend on one of sets of non-terminals.
4. Non-terminal productions can be recursive and mutually recursive.

To reflect associated relations between types, the type system needs to support, respectively, the notion of unit type, product types (or "records"), sum types (or "tagged unions") and recursive (or "inductive") types. All of these features are usually combined in a notion of algebraic data types (ADT).
More formally, the correspondence between non-terminal $V$ and associated type $\tau(V)$ is defined in the following way.

1. If $V$ has one production and it doesn’t contain non-terminals, i.e. $V \rightarrow \alpha_1 \alpha_2 \ldots \alpha_n$, then $\tau(V)$ is unit type.

2. If $V$ has one production of form $V \rightarrow \alpha_1 \alpha_2 V_2 \ldots \alpha_n V_n \alpha_{n+1}$, then $\tau(V) = \tau(V_1) \times \tau(V_2) \times \ldots \times \tau(V_n)$.

3. If $V$ has more than one production, i.e. $V \rightarrow \Omega_1 | \Omega_2 | \ldots | \Omega_n$, where $\Omega_k$ is a sequence $\alpha_1^{(k)} \alpha_2^{(k)} V_2^{(k)} \ldots \alpha_m^{(k)} V_m^{(k)} \alpha_{m+1}^{(k)}$, then $\tau(V) = \tau(V_1^{(1)}) \times \tau(V_2^{(1)}) \times \ldots \times \tau(V_1^{(n)}) + \ldots + \tau(V_1^{(n)}) \times \tau(V_2^{(n)}) \times \ldots \times \tau(V_1^{(n)})$.

The associated printer function $\Phi(v: \tau)$ returns concrete syntax representation of $v$. It can be inductively defined in the following way.

1. If $\tau(V)$ is unit type, then $\Phi(v: \tau(V)) = \alpha_1 \circ \alpha_2 \circ \ldots \circ \alpha_n$, where $\circ$ is a string concatenation operator.

2. If $\tau(V) = \tau(V_1) \times \tau(V_2) \times \ldots \times \tau(V_n)$, then $\Phi(v: \tau) = \alpha_1 \circ \Phi(v_1: \tau(V_1)) \circ \alpha_2 \circ \Phi(v_2: \tau(V_2)) \circ \ldots \circ \alpha_n \circ \Phi(v_n: \tau(V_n)) \circ \alpha_{n+1}$.

3. If $\tau(V) = \sigma_1 + \sigma_2 + \ldots + \sigma_n$, then $\Phi(v: \tau(V)) = \Phi(v': \sigma_i), i \in [1, n]$, where $v'$ is the value used to construct $v$ by applying $i$-th data constructor.

It is easily shown by structural induction on ADT value, that each application of printer function yields a sequence of terminal symbols that satisfies associated grammar. Therefore, syntactic correctness is ensured by type construction.

5. IMPLEMENTATION

The main result of this work is syngen (as for “syntax generator”) — a tool, that takes a context-free grammar and translate it to a source code generation framework implementation in some specified target programming language.

5.1 Language of implementation

syngen is implemented in Scheme programming language. The following reasoning advocates this choice.

Assume that syngen is written in some programming language $L$. To generate framework implementation in some target programming language $K$, it has to use a generator of $K$ source code. It is possible to acquire it by applying syngen to $K$ grammar, that should produce framework implementation in $L$. But then again, $L$ source code generator would be needed and so on.

To overcome this issue it is proposed to implement syngen in some Lisp dialect (Scheme is just the simplest of them). Lisp homoiconicity [10] allows to easily generate Lisp code within Lisp program. This way it is possible to immediately acquire a version of syngen that uses Lisp as target programming language. To provide support of another programming language (say, Java), it further needs to declare grammar of Java, produce Java source code generation framework and use it in Java-version of syngen. It is worth noting that there’s no need to implement the complete grammar, only a rather small subset of it, sufficient to express ADTs and declarations of printer functions, is needed.

This approach is similar to bootstrapping technique, widely used for compiler implementation.

Current implementation of syngen supports Scheme, Java and Haskell. As shown above, it is fairly easy to provide support of any other programming language.

5.2 Input grammar definition

Context-free grammar itself is completely sufficient (in theoretical sense) to describe most real-world languages, but in practice there is usually a separation between lexical and syntactic information of the language, which leads to separation of parsing process on lexical and syntactic stage. Lexical information is expressed as the set of tokens, each associated with regular expression. The grammar only operates on lexemes and does not consider their internal structure. Moreover, it is usually the case in source code generation, that the user already has atomic values (such as identifiers and numbers) that he doesn’t need to construct with the framework. For that reason, grammar declaration in syngen is split into definitions of tokens, and definition of non-terminals. Obviously, syntactic correctness of tokens (i.e. matching associated regular expression) can’t be easily checked on the type level, so it is checked in data constructors at run-time.

Syntax of grammar definition is mostly derived from the syntax of yacc [7] parser generator. Definition of a non-terminal consists of its name followed by a colon, a list of production rules separated by vertical bars, and a trailing semicolon. Production rule is a sequence of grammar symbols separated by whitespace.

Each production rule of input grammar should be annotated with the name of constructor for associated type. Name of constructor can either be stated explicitly in parenthesis before production rule, or omitted and inferred automatically from the context by the following rules.
1. If the non-terminal definition consists of only one production rule, the name of the non-terminal is assigned to the name of constructor.

2. If the first symbol of production is terminal, its value is assigned to the name of constructor.

3. Otherwise, syngen glues together values of terminals and names of non-terminals and assigns it to the name of constructor.

In case of name duplicate, the error will be displayed.

Appendix A shows the entire process of transforming context-free grammar to source code generation framework implementation. Appendix A.1 is a sample grammar of arithmetic expressions and one-arguments functions. Appendix A.2 is equivalent syngen definition, and Appendix A.3 is Haskell implementation of correspondent framework.

5.3 Embedding concerns
Design of produced source code generation framework is intentionally left as simple as possible, so that its implementation wouldn’t become an issue. Notion of ADT is fundamental for most statically typed functional programming languages, whereas in object-oriented languages, such as Java, it can be implemented with a combination of class inheritance and virtual dispatching.

Another concern is connected with atomic values in user’s program. Integers, float-point numbers and strings are appear quite often in produced source code, so it seems redundant to have to describe associated non-terminals and call associated data constructor. Therefore, those types are predefined in syngen and are not needed to be explicitly declared. Plain old integers, floats and strings, built in target programming language, can be used. Thus, in Appendix A.2 there is no definition for number non-terminal, because in this case it is more appropriate to use built-in integers.

6. FURTHER EXTENSIONS
After initial implementation, several extension were introduced to syngen grammar definition language. All of them are of sole purpose to improve user’s convenience.

6.1 Non-terminal closures
A common pattern in real world languages grammar is arbitrarily-size list of non-terminals (e.g. function arguments or statements in a block). It can be expressed recurrently, as follows.

\[
\text{arguments} : \text{argument} | \text{argument} ",", \text{arguments} ;
\]

However, this leads to cluttering the code, constructing list of arguments, since it needs to use nested arguments data constructor call. Instead, a notion of closures, similar to SDF [5] is proposed.

Four forms of lists of non-terminals are introduced, with and without a separator, and with zero or more (“Klonee closure”) or one or more elements (“positive closure”). Lists with Klonee closure are marked with trailing star symbol, and positive closure with plus sign. Optional separator is placed after caret symbol. Thus, the previous example of the argument list can be expressed as follows.

\[
\text{argument} : \text{argument} ^\ast ",", ^+ ;
\]

Appendix B.1 gives an example of JSON grammar, that use non-terminal closures to describe lists of values and entries of an object.

In order to absorb this extension, a small tuning of transformation algorithm is needed. It is quite natural to express associated type as a type of list, parameterized with the type, associated with the non-terminal alone. The notion of parameterized lists or arrays is fairly common in modern programming languages and question of embeddability is not the issue here.

The printer function for list of non-terminals simply glues together results of applying itself to elements of the list, delimited with provided separator.

6.2 Grammar constants
Grammar constants are user-defined sequences of terminals. Their purpose is to eliminate identical static parts of grammar definition, such as common keywords or delimiters. Besides that, they have more important application, explained in the following section.

Grammar constants are marked with dollar sign.

\[
\$\text{funstart} : \"\text{def}\" ;
\]

Besides terminals, grammar constants can use other grammar constants defined before. Obviously, recursive definitions are not allowed.

6.3 Indentation rules
Although, generated code is usually not intended to be read by a human, it is certainly useful to produce clean and readable code anyway, particularly, the code, that is correctly indented. Moreover, programming languages like Python or Haskell require correct indents in their syntax.

In order to remain context-free, the Python grammar introduces INDENT and DEDENT non-terminals to indicate transition to higher or lower level of indentation. The same idea is used in syngen.

Support of indentation is provided by assigning special side-effects to particular constants. Thus, \$\{ (\$\})\}, besides printing its value, increment (decrement) internal indent counter, and \$, print its value the number of times stored in the counter. Dollar sign in special constants can be omitted for brevity. Appendix B.2 gives an example of syngen definition of a tiny programming language with indents.

This extension needs tuning of printer function. Aside from printing value, it takes indentation counter and increment or decrement it in nested printer calls according to position of \$\{ and \$\}.

7. CONCLUSIONS

This paper makes an effort to provide a tool, that eliminates the need of code templates in source code generation applications. Instead, it is proposed to automatically produce source code generation framework from given grammar definition. This approach increases reliability and maintainability of the code.

As mentioned before, grammar definition is not required to fully cover syntax of the language. This allows to implement source code generator incrementally, introducing new syntactic structures when they are needed. Adding new non-terminals along with adding new productions for existing non-terminals doesn’t affect the interface of the framework.

Grammar definition, employed for language specification, is allowed, in spite of parsing grammar, to be ambiguous, as the ambiguity is resolved by construction of ADT value. Moreover, this property can be utilized in the following way. Consider the grammar of function signatures of Java programming language. It would probably include non-terminals representing presence or absence of static keyword, access modifiers and return types. If functions of certain signature appear frequently in the code (e.g. `public void test ()` for JUnit tests), it might be tedious to produce the same syntactic structure over and over again. Instead, one would prefer to define new non-terminal

\[
\text{testfun} : \text{"public" \ "void" \ funname \ "()" ;}
\]

that can be produced with a single value.

**syngen** is intended to be used in compilers, preprocessors and other kind of tools that need to describe complex grammatical structures. On the other hand, it might be suboptimal to use **syngen** for the tools that mostly produce static fragments of code with single inclusions of atomic values, as the overhead for proper grammar description will probably outweigh other benefits. In such cases, code templates remain to be more fortunate choice.

8. FUTURE WORK

**syngen** was developed in assumption, that produced language can be defined with context-free grammar. However, for certain real-world languages, like XML with arbitrary scheme, this is not true. For this particular case a possible extension of input grammar is the ability to define whether or not some non-terminals have identical productions.

As mentioned earlier, input grammar is allowed to be ambiguous, because it is used for producing and not parsing. A drawback of this property is a need to constantly surround infix operators with parenthesis. This can affect readability of the output code. Another direction of future work on **syngen** is introducing the concept of priority of infix operators to grammar definition. N. Ramsey [12] introduced an algorithm of automatically placing parenthesis in concrete syntax of code fragments with priority-defined infix operators. It can be incorporated in future versions of **syngen** implementations.

9. REFERENCES


APPENDIX
A. FRAMEWORK GENERATION
A.1 Sample grammar
\[
\begin{align*}
\text{expr} & \rightarrow \text{number} \mid \text{operation} \mid \text{funcall} \\
\text{operation} & \rightarrow \text{expr} + \text{expr} \mid \text{expr} \times \text{expr} \\
\text{funcall} & \rightarrow \text{variable} (\text{expr}) \\
\text{number} & \rightarrow 0 \mid 1 \mid 2 \ldots \mid 9 \\
\text{variable} & \rightarrow (a \mid \ldots \mid z) +
\end{align*}
\]

A.2 Equivalent syngen definition
\[
\begin{align*}
\text{expr} : (\text{number}) \, \text{integer} \mid \text{operation} \mid \text{funcall} ; \\
\text{operation} : (\text{plus}) \, \text{expr} \, " + " \, \text{expr} \mid (\text{times}) \, \text{expr} \, " \ast " \, \text{expr} ; \\
\text{funcall} : (\text{function}) \, \text{variable} \, " ( " \, \text{expr} \, " ) " ; \\
\text{variable} : / [a-z]+ / ;
\end{align*}
\]

A.3 Framework implementation
module Sample (Expr, number, operation, funcall, Operation, plus, times, Funcall, function, Variable, variable) where
import Text.Regex.Posix
data Expr = Number Integer | Operation Operation | Funcall Funcall
number = Number
operation = Operation
funcall = Funcall
data Operation = Plus Expr Expr | Times Expr Expr
plus = Plus
times = Times
data Funcall = Function Variable Expr
function = Function
data Variable = Variable String
variable s | s = "[a-zA-Z]+$" = Variable s
| otherwise = error "Type error"
class Printer a where
pr :: a \rightarrow String
instance Printer Expr where
pr (Number i) = show i
pr (Operation o) = pr o
pr (Funcall f) = pr f
instance Printer Operation where
pr (Plus e1 e2) = pr e1 ++ "+" ++ pr e2
pr (Times e1 e2) = pr e1 ++ "\ast " ++ pr e2
instance Printer Funcall where
pr (Function v e) = pr v ++ "(" ++ pr e ++ ")"
instance Printer Variable where
pr (Variable s) = s

B. SAMPLES OF SYNGEN DEFINITIONS
B.1 JSON
value : "null" | "true" | "false" | string
| (array) "[" value", "s "]" | (object) "{" entry", "s "}"
; 
entry : string ":" value ;

B.2 Tiny programming language with indents
{ : "\n" ;
id : /[a-zA-Z-\[-Za-z0-9]+\]/ ;
unit : statement* ;
- statement "ln"
  : "def" id "(" id", "s ")" ( statement+ )
  | "return" expression
  | "if" expression ( statement+ )
 ;
expression
  : (ref) id
  | (int) integer
  | (op) expression operator expression
  | (call) id "(" expression", "s ")"
 ;
operator
  : (equal) "=="
  | (plus) "+"
  | (minus) "-"
  | (mult) "\ast "
 ;