Implementing a Parser for *Elements of Programming*

Carla Villoria  Gabriel Dos Reis

Department of Computer Science and Engineering
Texas A&M University

TAMU-CSE-2009-10-1

October 2009

Abstract
This document is the first in a series that reports on an ongoing effort to implement Liz, an interpreter for structured generic programming with C++-like language. We present a parser for Liz. It handles all of the programs in *Elements of Programming*, and beyond. We also include grammar extensions to express axioms and concepts in code.

1 Introduction

Alexander Stepanov and Paul McJones recently published a magnificent book [5] on structured generic programming titled *Elements of Programming*. They showcase programming as a mathematical activity, a wonderful journey in the land of simplicity and generality. Their approach makes essential use of axioms (and more generally properties) and concepts, i.e. collections of syntactic, semantics, and complexity requirements on datatypes and operations. In spite of this reliance on concepts, all of their codes is compilable as almost C++03 [4] program fragments. That is accomplished essentially by use of a few simple macros. In particular, the requires “keyword” is actually a C99 variadic macro defined [5, Appendix B.2] to ignore its arguments. Consequently, one of the benefits of concepts — turning informal descriptions into codes so that they can be verified and used for type checking template definitions and uses — is not realized.

This document is the first in a series of reports on an ongoing effort to implement Liz, an interpreter that offers linguistic support for the style of structured generic programming advocated in *Elements of Programming*. Here we present a grammar of the language supported by Liz and its implementation. It is a regular subset of C++, augmented with axioms [2] and concepts [1, 3]. The parser
discussed in this report is a recursive descent parser, written in Standard C++, using parser combinator technology.

2 Parser Combinators

The grammar and semantics sketch for the subset of C++ used in Elements of Programming is described on less than ten pages [5, pp 233–241]. This terse description relies on knowledge available elsewhere in the literature. The grammar was designed to be almost context-free — this is to be contrasted with Standard C++ grammar which necessitates semantics processing. As explained in [5, p. 239], the only exception is the usual case where a template specialization is explicitly named: an identifier followed by the less-than symbol followed by an additive expression is a valid production (a relational expression). The ambiguity is resolved by checking whether the identifier names a template. This is the only case where context matters during parsing.

The grammar put forward in [5] uses the extended Backus–Naur form advocated by Niklaus Wirth [6]. The essential feature of EBNF is the introduction of an explicit iteration construct \{a\} to stand for \(\epsilon|a|aa|aaa|\ldots\). EBNF also introduced optionality of a rule a by \([a]\). We follow suit. We also introduce shortcuts for some syntactic patterns that appear over and over again. All of these shortcuts are expressible in the EBNF at the expense of repetition and poor abstraction of commonalities. We will use usual alphanumeric identifiers in functional notation for those meta constructions. To keep the notation uniform, we also introduce a functional notation for the essential meta constructs of EBNF:

- **ZeroOrMore(a)** is EBNF’s iteration \{a\} of rule a.
- **OneOf([a_1, a_2])** is EBNF’s choice \(a_1|a_2\). More generally **OneOf([a_1, a_2, \ldots, a_n])** is \(a_1|a_2|\ldots|a_n\) in EBNF notation.

There are places in the grammar where a production is enclosed in “brackets”; for example:

- function argument lists are enclosed in parenthesis
- compound statements are lists of statements enclosed in curly braces
- indexing into an array is an expression in square brackets
- template argument lists are expressions enclosed in angle brackets

All of these instances of bracketing are captured by the combinator Enclosed which can be instantiated with two arguments: the first is a rule and the second is a bracket token. The instantiation is a rule consisting of the terminal bracket followed by rule, followed by a matching bracket:
Enclosed\((\text{rule, bracket})\) = \\
\text{bracket rule Closer(bracket)}

Parenthesized\((\text{rule})\) = \\
\text{Enclosed(\text{rule, OPEN_PAREN})}

Bracketed\((\text{rule})\) = \\
\text{Enclosed(\text{rule, OPEN_BRACKET})}

Braced\((\text{rule})\) = \\
\text{Enclosed(\text{rule, OPEN_BRACE})}

Angled\((\text{rule})\) = \\
\text{Enclosed(\text{rule, LT})}

The meta combinator Closer is defined by cases on “bracket” tokens:

\begin{align*}
\text{Closer(OPEN_PAREN)} & = \text{CLOSE_PAREN} \\
\text{Closer(OPEN_BRACKET)} & = \text{CLOSE_BRACKET} \\
\text{Closer(OPEN_BRACE)} & = \text{CLOSE_BRACE} \\
\text{Closer(LT)} & = \text{GT}
\end{align*}

Another combinator that is useful to describe the syntactic structure of expressions is LeftAssociative:

\begin{align*}
\text{LeftAssociative(\text{rule, ops})} & = \\
& \text{rule ZeroOrMore(OneOf(ops) rule)}
\end{align*}

This combinator describes binary expressions where the operator associates to the left, as is traditionally the case for additive expressions. A dual combinator, RightAssociative is also defined. Finally, we also used comma-separated items:

\begin{align*}
\text{CommaSeparatedList(\text{rule})} & = \\
& \text{rule ZeroOrMore(COMMA rule)}
\end{align*}

3 Liz’s Grammar

3.1 Toplevel statements

\begin{verbatim}
toplevel = 
   OneOf(template, concept, axiom, structure, procedure, statement)
\end{verbatim}

The language described in Appendix B.1 of Elements of Programming does not include any production for the toplevel. In particular, no description is given for the structure of a program. This is probably because the book concentrates only on algorithms, e.g. program fragments as opposed to complete applications. However, after the initial completion of this work, Sean Parent indicated that the toplevel was envisioned to consist of \text{template, structure} and \text{procedure}. We included \text{statement} at the toplevel because Liz is primarily designed for interactive use and we did not find it a good design to invent an
entirely new, different language for interactive uses. We also note that the inclusion of statement at toplevel (which includes simple statements) provides a convenient way to work around the C preprocessor \texttt{#include} directive; e.g.

we write

\begin{verbatim}
import("eop.h");
\end{verbatim}

instead of

\begin{verbatim}
#include "eop.h"
\end{verbatim}

We are loath to implement a C preprocessor, and we would prefer a compile-time evaluation mechanism for the programming style advocated in \textit{Elements of Programming}.

The grammar for \textit{axiom} and \textit{concept} definitions are not in the \textit{Elements of Programming}. They are our proposal for Liz. The \textit{concept} production differs from past concept proposals [1, 3]. The rule for \textit{axiom} also differs from what was described in [2]. It particular, it contains explicit support for universal quantification, instead of relying on indirect encoding through Skolemization.

### 3.1.1 Templates

\begin{verbatim}
template =
    TEMPLATE Angled(Optional(parameter_list))
    Optional(constraint) OneOf(structure, procedure, specialization)
\end{verbatim}

\begin{verbatim}
constraint =
    REQUIRES condition
\end{verbatim}

\begin{verbatim}
condition =
    Parenthesized(expression)
\end{verbatim}

\begin{verbatim}
specialization =
    STRUCT structure_name Angled(additive_list)
    Optional(structure_body) SEMICOLON
\end{verbatim}

The grammar for template declarations is unchanged from [5].

### 3.1.2 Concepts

\begin{verbatim}
concept =
    CONCEPT identifier Parenthesized(parameter_list)
    Braced(ZeroOrMore(concept_clause))
\end{verbatim}

\begin{verbatim}
concept_clause =
    OneOf(universal, axiom, procedure)
\end{verbatim}

For example, here is a definition of the \textit{HomogeneousFunction} concept [5, p. 12] is written in Liz as
concept HomogeneousFunction(Function F) {
    Arity(F) > 0;
    forall(int i, int j) i < Arity(F) and j < Arity(F) =>
        InputType(F, i) == InputType(F, j)
    Regular Domain(HomogeneousFunction T) {
        return InputType(T, Φ);
    }
}

3.1.3 Axioms

axiom =
    AXIOM  identifier  Optional(Parenthesized(parameter_list))  Braced(universal)

universal =
    ZeroOrMore(FORALL  Parenthesized(parameter_list))  proposition

proposition =
    RighAssociative(expression, [IMPLIES])  SEMICOLON

The axiom constructs lets programmers express properties in program. For the example the regular Unary_function [5, p. 14] property is expressed as

axiom regular Unary_function(UnaryFunction F) {
    forall(F f1, F f2) forall(Domain(F) x1, Domain(F) x2)
        f1 == f2 and x1 == x2 => f1(x1) == f2(x2);
}

3.1.4 Structures

structure =
    STRUCT  structure_name  Optional(structure_body)  SEMICOLON

structure_name =
    identifier

structure_body =
    Braced(ZeroOrMore(member))

member =
    OneOf(data member, constructor, destructor, assign, apply, index, typedef)

data member =
    expression  identifier
        Bracketed(OPEN_BRACKET expression  CLOSE_BRACKET)  SEMICOLON

constructor =
    structure_name  Parenthesized(Optional(parameter_list))
        Optional(COLON  CommaSeparatedList(initializer))  body
3.1.5 Procedures

procedure =
expression procedure_name Parenthesized( Optional( parameter_list ))
OneOf( body, SEMICOLON )

procedure_name =
OneOf( identifier, operator )

operator =
OPERATOR OneOf( [ DOUBLE_EQ, LT, PLUS, MINUS, STAR, SLASH, PERCENT ] )

parameter_list =
CommaSeparatedList( parameter )

parameter =
expression Optional( identifier )

body =
Compound

Note that in Elements of Programming, the type void was not considered an expression. However, we believe that a uniform treatment of type expressions benefits from viewing void just as any other type expression.
3.1.6 Statements

\[
\text{statement} = \begin{aligned}
&\text{Optional(}\text{identifier COLON)} \\
&\text{OneOf(}} (\text{simple\_statement, assignment, construction, control\_statement, typedef})
\end{aligned}
\]

\[
\text{simple\_statement} = \\
\text{expression SEMICOLON}
\]

\[
\text{assignment} = \\
\text{expression EQ expression SEMICOLON}
\]

\[
\text{construction} = \\
\text{expression identifier Optional(initialization) SEMICOLON}
\]

\[
\text{initialization} = \\
\text{OneOf(}} (\text{Braced(expression\_list), EQ expression})
\]

\[
\text{control\_statement} = \\
\text{OneOf(}} \text{return, conditional, switch, while, do, compound, break, goto)
\]

\[
\text{switch} = \\
\text{SWITCH condition Braced(ZeroOrMore(case))}
\]

\[
\text{case} = \\
\text{CASE expression COLON ZeroOrMore(statement)}
\]

\[
\text{while} = \\
\text{WHILE condition statement}
\]

\[
\text{do} = \\
\text{DO statement WHILE condition}
\]

\[
\text{compound} = \\
\text{Braced(ZeroOrMore(statement))}
\]

\[
\text{break} = \\
\text{BREAK SEMICOLON}
\]

\[
\text{goto} = \\
\text{GOTO identifier SEMICOLON}
\]

\[
\text{typedef} = \\
\text{TYPEDEF expression identifier SEMICOLON}
\]

Note that the production \textit{switch} appears twice in the first printing of Elements of Programming. In Liz, we consider a \textit{typedef} statement as obsolete because the same effect can be had with using a “variable”-like definition, e.g.

\[
\text{typename size\_t = unsigned;}
\]
thanks to the fact that Liz considers `typename` as a built-in basic type.

### 3.2 Expressions

- **expression:** `LeftAssociative(conjunction, [DOUBLE_BAR, OR])`
- **conjunction:** `LeftAssociative(equality, [DOUBLE_AMPER, AND])`
- **equality:** `LeftAssociative(relational, [EQ, NEQ])`
- **relational:** `LeftAssociative(additive, [LT, LE, GT, GE])`
- **additive:** `LeftAssociative(multiplicative, [PLUS, MINUS])`
- **multiplicative:** `LeftAssociative(prefix, [STAR, SLASH, PERCENT])`
- **prefix:** `MaybeOneOf(MINUS, EXCLAMATION, CONST) postfix`
- **postfix:** `primary ZeroOrMore(OneOf(DOT identifier, Parenthesized(Optional(expression_list)), Bracketed(expression)), AMPER)`
- **primary:** `OneOf(literal, identifier, Parenthesized(expression), basic_type, template_name)`
- **template_name:** `identifier Angled(additive_list)`
- **basic_type:** `OneOf(BOOL, CHAR, INT, DOUBLE, STRING, TYPENAME)`

### 4 Implementation

In its current implementation, there is a one to one correspondence between the grammar described in previous section and actual C++ code, thanks to the systematic use of parser combinators.

### 5 Related Work

After initial completion of this work, we were informed by Sean Parent and Paul McJones of a parser for the elements, also written in C++, that was developed
by Parent. Our understanding is that Parent’s parser was more extensive that
the description given in the Elements of Programming. However our parser ap-
pear to differ from Parent’s in that it offers extensions to expression axioms and
concepts directly in code.

Acknowledgment

We would like to thank Paul McJones and Sean Parent for helpful comments
during the implementation of this parser.

References

[1] Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ Concepts. In Con-
ference Record of POPL ’06: The 33th ACM SIGPLAN-SIGACT Symposium on
Principles of Programming Languages, pages 295–308, Charleston, South Car-
olina, USA, 2006.

[2] Gabriel Dos Reis, Bjarne Stroustrup, and Alisdair Meredith. Axioms:
Semantics Aspects of C++ Concepts. Technical Report N2887=09-0077,

Dos Reis, and Andrew Lumsdaine. Concepts: Linguistic Support for
Generic Programming in C++. In OOPSLA ’06: Proceedings of the 21st an-
nual ACM SIGPLAN conference on Object-Oriented Programming Languages,
Press.


Wesley, 2009.

[6] Niklaus Wirth. What Can We Do About the Unnecessary Diversity of No-