PLCC: A Programming Language Compiler Compiler

Timothy Fossum
Computer Science Department
SUNY College at Potsdam
Potsdam, NY 13676
fossumtv@potsdam.edu

ABSTRACT

This paper describes PLCC, a compiler-compiler tool to support courses in programming languages, compilers, and computational theory. This tool has proven to be useful for implementing interpreters, building compilers, and creating parsers for context-free languages.

PLCC is a Perl program that takes an input file that specifies the tokens, syntax, and semantics of a language and that generates a complete set of Java files that implement the semantics of the language. PLCC stands for "Programming Language Compiler-Compiler".

PLCC is not intended to be a production-quality tool. Rather, it supports understanding and implementing the essential elements of lexical analysis, parsing, and semantics without having to wrestle with the complexities of dealing with "industrial-strength" compiler-compiler tools. Students quickly learn how to write PLCC "grammar" files for small languages that have straightforward syntax and semantics and use PLCC to build Java-based parsers, interpreters, or compilers for these languages that run out-of-the-box.

Input to PLCC is a text file with a token definition section that defines language tokens as simple regular expressions, a syntax section that specifies the grammar rules of an LL(1) language as simple Backus-Naur Form (BNF) productions, and a semantics section that defines the language semantics as Java methods.

PLCC generates a set of Java source files that are entirely self-contained and that import only standard elements of `java.util` in JDK5 and above. For testing purposes, PLCC generates a read-eval-print loop that (1) reads standard input, (2) scans, parses, and evaluates the input, and (3) prints the evaluation to standard output.

Categories and Subject Descriptors

D.3.4 [**Software**]: Programming Languages—*processors*

SIGCSE'14, March 3–8, 2014, Atlanta, GA, USA.

http://dx.doi.org/10.1145/2538862.2538922.

General Terms

Languages

Keywords

Compiler-compiler, parser, interpreter, syntax, semantics

1. INTRODUCTION

My decision to write PLCC was inspired by the book *Essentials of Programming Languages* (EOPL) by Friedman, Wand, and Haynes[4]. I had used EOPL for more than ten years to teach an upper-level course in Programming Languages. Since Java is the principal language our students take in introductory courses, I found the time required for them to learn Scheme, the implementation language used in EPOL, ate into the time I wanted to cover other topics in my Programming Languages class.

What I liked about EOPL was the emphasis on writing simple interpreters for languages that grow incrementally in complexity as the course progresses. Many of these languages are purely functional, giving students much-needed exposure to a different programming paradigm. My *ITiCSE* paper[2] describes how I use the EOPL approach to define a language with classes as first class objects, which is based on material in Chapter 5 (Objects and Classes) of EOPL.

EOPL comes with a Scheme tool set called `sllgen`, a parser-generator written in Scheme. Input to `sllgen` consists of specifications (represented as Scheme lists) of a language's lexical structure and grammar. `Sllgen` uses the lexical specification to generate a scanner for the language and uses the grammar specification (which must be LL(1)[1, Chapter 5]) to generate a set of Scheme "datatypes" that implement the elements of a parse tree for a program in the language. The `sllgen`-generated recursive descent parser returns an instance of the datatype for the start symbol of the grammar; this instance is the root of the parse tree for the input program in the language. An `sllgen`-generated `read-eval-print` loop reads a program in the language, tokenizes and parses it, evaluates the resulting parse tree based on semantics provided by the implementer in an `eval` procedure, and prints the result.

The "datatypes" generated by `sllgen` are akin to Pascal's *variant records*[6]. When two or more grammar rules have the same left-hand-side nonterminal, the `sllgen` parser chooses the correct grammar rule to apply based on the current token (using the LL(1) properties of the language) and returns an instance of the nonterminal's datatype that identifies what grammar rule was used in the parse. Scheme

procedures like `eval` that perform semantic actions on these datatypes must use a `cases` construct – like a `switch` in Java or C – to identify the particular instance of the datatype and to carry out an appropriate semantic action on that instance. Writing `eval` code using a `cases` construct can be messy. Each of the grammar rules having the same left-hand-side nonterminal requires a `cases` entry to code the `eval` behavior of a particular variant of the datatype. This makes coding cumbersome, and it doesn't facilitate a clean physical or conceptual separation of responsibilities to handle the `eval` semantics that the different right-hand-sides represent.

I observed that processing the variants of a datatype could be accomplished through dynamic dispatch in an object-oriented language. Instead of using a `cases` construct to `eval` a Scheme datatype representing a node in a parse tree, I could use inheritance to dispatch an `eval` method call on the Java object that represents the particular node in the parse tree. Dynamic dispatch has the advantage that the `eval` code for the object representing a node in the parse tree node is unique to the object.

I also wanted to represent the lexical and grammar specifications of a language in a way that is more straightforward than using Scheme lists. In particular, I wanted to use *regular expressions*[3] to define the tokens in the language and to use conventional Backus-Naur Form (BNF) style for specifying the grammar rules. Finally, I wanted the entire language specification to be monolithic, with the lexical, grammar, and semantic specifications all in one file.

I liked being able to implement the increasingly complex sequence of languages as given in the EOPL book. I wanted to ensure that the PLCC project would allow me to process these languages as I had done using Scheme and `sllgen`.

My goals in this project were thus to:

- Target Java as the implementation language
- Use a single file to specify the language's lexical, grammar, and semantic structure
- Use regular expressions to specify the lexical structure of a language
- Use BNF to define the grammar rules of a language
- Generate recursive-descent parsing code that is easy to read and understand
- Make a clear separation of semantics from the syntax
- Make use of dynamic dispatch to implement semantic actions corresponding to different grammar rules having the same left-hand-side nonterminal
- Be capable of handling language examples in EOPL
- Use standard Perl and Java to implement the project

After building PLCC and using it in my Programming Languages course based on EOPL, I found that PLCC would also serve as a framework for teaching a course on compilers. For my Compilers course, I replace the simple PLCC token scanner based on regular expressions with a separately built token scanner based on a finite-state machine (also using an object-oriented approach) that can handle, for example, comments that cross line boundaries. When used to generate a compiler, PLCC handles target code generation in the same way it handles evaluation semantics for an interpreter, except that the "value" of a program in the source language is a generated program in the target language (I use assembly language), and the read-eval-print loop is replaced by a read-parse-generate method.

A bare-bones PLCC specification without any evaluation semantics (other than returning "pass" or "fail") can be used to generate parsers for many context-free languages encountered in a Computational Theory class. It's easy to construct a simple specification file for a language with a small number of tokens and grammar rules that PLCC can turn into a read-eval-print loop to check for membership in the language.

2. THE PLCC TOOL

2.1 Lexical specifications

The EOPL `sllgen` toolkit defines the lexical specification of a language in terms of a Scheme list. The following is a lexical specification in `sllgen` that skips whitespace and comments (beginning with the '`%`' character) and that identifies patterns for variables (`var`) and numeric literals (`lit`):

```
(define the-lexical-spec
  '((whitespace
      (whitespace) skip)
    (comment
      ("%" (arbno (not #\newline))) skip)
    (var
      (letter (arbno (or letter digit))) var)
    (lit
      (digit (arbno digit)) lit)))))
```

In PLCC, I represent these patterns as regular expressions. Here is the the same lexical specification written in PLCC, which I regard as more straightforward. This approach also gives students the opportunity to learn about regular expressions, which can prove to be useful to them as they pursue a computing-related career.

```
skip WHITESPACE '\s+'
skip COMMENT '%.*'
LIT '\d+'
VAR '[a-zA-Z]\w*'
```

PLCC assumes that tokens do not cross line boundaries. This means that a pattern such as '`%.*`' will terminate at the end of the current line. This assumption makes token processing simpler, but it does mean that the PLCC-generated scanner cannot handle multi-line skips or tokens. To allow for more general lexical analysis behavior, PLCC can be directed to skip generating the scanner-related classes, allowing the user to provide an externally written scanner. As noted above, I use this in my compiler class, where we spend time implementing a stand-alone scanner.

`Sllgen` gathers reserved words from the grammar rules, so the following `sllgen` grammar rule would result in defining tokens corresponding to '`if`', '`then`', and '`else`':

```
(exp
  ("if" exp "then" exp "else" exp)
  if-exp)
```

In PLCC, these tokens need to be defined explicitly in the lexical specification section:

```
IF 'if'
THEN 'then'
ELSE 'else'
```

The PLCC-generated scanner skips input that matches the lexical `skip` definitions. It then returns the next input token by examining all of the token definitions, in the order given in the specification file, and determining which ones match one or more characters of the current input. The scanner returns the token with the longest match: among matches of the same length, it returns the token corresponding to the first definition it encounters. This means that the lexeme '`if`' would be returned as an IF token, not a VAR token, with the following token specifications:

```
skip WHITESPACE '\s+'
skip COMMENT '%.*'
IF 'if'
THEN 'then'
ELSE 'else'
LIT '\d+'
VAR '[a-zA-Z]\w*'
```

When I ask students to add a new reserved word such as '`while`' to the token specifications, some will add it *after* the VAR specification, so '`while`' would be regarded as a `VAR` instead of a `WHILE`: both patterns match the string '`while`', and both matches have have the same length, but `VAR` comes first. This provides a good learning opportunity: a test program that should work with a `while` will not parse properly if `while` is treated as a `VAR`. Some students who make this error ask why their solutions don't work, and they have an "aha" experience when they discover – often with my help – why not, and how to fix it by simply moving a line in the token specifications. Others who submit incorrect solutions without even writing test programs are greeted with a pointed query asking if they tested their solutions.

2.2 Grammar Rules

`Sllgen` grammar rules are given as a Scheme list. Here is an example subset of grammar rules for expressions in one of the EOPL languages:

```
(exp (lit) lit-exp)
(exp (var) var-exp)
(exp
  ("if" exp "then" exp "else" exp)
  if-exp)
(exp
  ("let" (arbno var "=" exp) "in" exp)
  let-exp)
(exp
  ("proc" "(" (separated-list var ",") ")" exp)
  proc-exp)
```

The first two grammar rules above can be written in PLCC as follows:

```
<exp>:LitExp ::= <LIT>
<exp>:VarExp ::= <VAR>
```

The `arbno` construct in `sllgen` corresponds to a "Kleene star" repetition construct in Extended BNF. PLCC does not allow a Kleene star operator to be used in the middle of the right-hand-side of a grammar rule. Instead, a separate grammar rule construct, introduced by '`**=`', must be used to identify a rule where the entire right-hand-side can appear zero or more times. For example, the `(arbno var "=" exp)` construct can be written as a separate rule in PLCC as follows:

```
<letDecls> **= <VAR> EQUALS <exp>
```

Armed with this, the `sllgen` grammar `let-exp` rule can be written in PLCC as shown here:

```
<exp>:LetExp ::= LET <letDecls> IN <exp>
<letDecls>   **= <VAR> EQUALS <exp>
```

The `separated-list` construct in `sllgen` identifies a rule where items can appear zero or more times – similar to `arbno` – but the items must be separated by a specified token if there are more than one of them. This is used, in `proc-exp` for example, when specifying function parameters as a comma-separated list:

```
(separated-list var ",")
```

PLCC uses '`**=`' to introduce such a construct, with the separator token appearing at the end preceded by a `+` sign:

```
<formals> **= <VAR> +COMMA
```

A `proc-exp` in PLCC will then become

```
<exp>:ProcExp ::= PROC
    LPAREN <formals> RPAREN <exp>
<formals>     **= <VAR> +COMMA
```

(Note that the first two lines above appear folded to fit the column width. In the PLCC file, these would appear on one line.)

2.3 Classes generated from grammar rules

PLCC generates a Java class for each left-hand-side nonterminal given in the grammar rules. Nonterminals in the grammar rules section must begin with a lower-case letter and can be followed by any number of additional letters, digits, or underscores. The name of the Java class generated by PLCC is the same as the name of the nonterminal, except with its first letter converted to uppercase. From the nonterminals `<exp>`, `<letDecls>` and `<formals>` PLCC generates the classes `Exp`, `LetDecls` and `Formals`.

When a language has two or more grammar rules with the same left-hand-side nonterminal, the left-hand-side nonterminal class is declared as `abstract`, and subclasses are created for each of these grammar rules. The name of a particular subclass is specified in the grammar rule by the class name that follows a colon '`:`' after the nonterminal. For example, the following grammar rules

```
<exp>:LitExp ::= <LIT>
<exp>:VarExp ::= <VAR>
```

define classes named `LitExp` and `VarExp` that extend the abstract class `Exp`. PLCC ensures that there is a one-to-one correspondence between the generated non-abstract class names and the grammar rules.

2.4 Class fields

As described above, every grammar rule line defines a unique non-abstract Java class. Each such class has a number of `public` fields corresponding to the items on the right-hand-side of the rule. Only those right-hand side items that appear in angle brackets '`<...>`' have fields defined in the class. If the item is a nonterminal such as `<exp>`, the corresponding field is named `exp` and has type `Exp`. If the item is a terminal such as `<VAR>`, the corresponding field is named `var` and has type `Token`. The class has a single constructor that assigns its arguments to these fields.

For example, from the grammar rule

```
<exp>:LetExp ::= LET <letDecls> IN <exp>
```

PLCC generates a `LetExp` class having two fields, a constructor, and a static `parse` method:

```
// <exp>:LetExp ::= LET <letDecls> IN <exp>
public class LetExp extends Exp {

    public LetDecls letDecls;
    public Exp exp;

    public LetExp (LetDecls letDecls,
                   Exp exp) {
       this.letDecls = letDecls;
       this.exp = exp;
    }

    public static LetExp parse(Scan scn) {
        scn.match(Token.Val.LET);
        LetDecls letDecls = LetDecls.parse(scn);
        scn.match(Token.Val.IN);
        Exp exp = Exp.parse(scn);
        return new LetExp(letDecls, exp);
    }
...

}
```

The static `LetExp.parse` method returns an instance of this class by processing, in order, the items in the right-hand-side of the grammar rule: matching the terminal `LET`, calling the `LetDecls.parse` method, matching the terminal `IN`, and calling the `Exp.parse` method. The resulting `letDecls` and `exp` values are used to construct and return a `LetExp` object.

PLCC declares its generated class fields to be `public`. While this practice can be considered unsafe, it makes coding semantic methods more straightforward.

A Java class generated by a grammar rule with repetitions – one that uses the `**=` repetition construct – can have zero or more field instances corresponding to items on its right-hand-side. For such a class, the values are collected into `ArrayList` fields whose names are the same as for a non-repeating rule, with the string '`List`' appended. For example, the following grammar rule

```
<letDecls> **= <VAR> EQUALS <exp>
```

generates a Java class named `LetDecls` having a field named `varList` of type `ArrayList<Token>` and a field named `expList` of type `ArrayList<Exp>`. A successful call to the `parse` method on this class returns an instance of a `LetDecls` object: the `varList` will be populated with a number of `Token` objects, and the `expList` will be populated with the same number of `Exp` objects. Similar remarks apply to repeating rules with a separator.

Both of the repeating rules can be replaced with suitably chosen recursive grammar rules: PLCC makes such replacements internally, but only to check for LL(1). However, there are significant advantages to using repeating rules:

- The resulting parse trees for grammars using repeating rules are shallower than those using recursive grammar rules, since the `ArrayList`s flatten out the parse tree.
- The `parse` methods for a class defined by a repeating rule can employ a loop instead of recursive calls, resulting in run-time improvements in space and time. The same remark applies to methods that implement repeating rule semantics.
- Repeating rules are easier to read and understand than their recursive counterparts: there are fewer productions, and it's easy to spot the repeating parts.
- The `ArrayList` fields in a repeating rule conveniently package the repeating elements of a parse in a way that is more direct and compact than spreading them out in the parse tree. Processing these `ArrayList`s to carry out semantic actions is straightforward using iterators.

2.5 Parsing

Parsing a program that conforms to a PLCC grammar specification is easy: call the `parse` method on the class generated by the start symbol of the language, which is always the first left-hand-side nonterminal appearing in the grammar rules. The `parse` method is static in all of the PLCC-generated classes. Each `parse` method is passed a `Scan` object (see Section 3 below) that delivers tokens for parsing. For an abstract class such as `Exp`, the `parse` method returns a parsed instance of one of its subclasses: the current input token determines the appropriate grammar rule to apply (based on the LL(1) property of the grammar) which in turn determines the appropriate subclass to instantiate. For a non-abstract class such as `LetDecls`, the `parse` method returns an instance of the class itself.

When the top-level parse completes successfully (exceptions can occur when there is a syntax error or when the `Scan` class is unable to deliver a token) the result is an instance of the Java class corresponding to the start symbol. In many EOPL languages, this is an instance of the `Program` class.

The following Java code represents the essence of PLCC-generated scanning and parsing, returning an instance of the `Program` class, the class associated with the start symbol `<program>`:

```
Program.parse(new Scan(System.in));
```

2.6 Semantics

The PLCC-generated read-eval-print loop parses a program in the language by calling the static `parse` method on the Java class generated by the start symbol of the language. This is done *before* any semantics are applied to the resulting parse. Thus the entire program is scanned and parsed prior to carrying out any semantic actions.

The default PLCC semantics of a program is to print the Java `String` value of the parse. In other words, the entirety of the default PLCC lexical analysis, parsing, and semantics is embodied in the following one-liner (folded for your viewing pleasure):

```
System.out.println(
  Program.parse(
    new Scan(System.in)
  )
);
```

In the absence of overriding the `toString` method of a `Program` object, the above statement will produce something like

```
Program@768965fb
```

which simply says that the default semantics of this program is a `Program` object.

In its simplest form, implementing the non-default semantics of a PLCC language consists of overriding the default `toString` method in the start symbol class. Here is a grammar fragment (the tokens specifications are similar to those given above) for a simple language:

```
<program>    ::= <exp>
<exp>:LitExp ::= <LIT>
<exp>:VarExp ::= <VAR>
```

To override the default `toString` behavior in the `Program` class, we create a code fragment that defines a `toString` method and associate it with the `Program` class as follows:

```
Program
%%{
    public String toString() {
        return exp.eval();
    }
%%}
```

The first line, beginning with `Program`, identifies the Java class whose source code will be modified, and the items between the lines `%%{` and `%%}` will be added to the code already in the `Program.java` source file. (In the above example, recall that `exp` is a field in the `Program` class that is populated by its `parse` method. The `eval` method will be defined in the subclasses of the `Exp` class.) This code, and all other code that defines the language semantics of the grammar rules, appears in the PLCC language specification file following the grammar rules for the language, after a line with a single '`%`'.

An entire Java class source file can be created in the semantics section by naming the class (as long as it is different from the PLCC-generated classes) and including the code to be inserted into the class between `%%{` and `%%}` lines. This is useful when Java classes other than those automatically generated by PLCC are needed to implement language semantics.

Continuing the example above, the `eval` semantics for a `LitExp` might be given as follows:

```
LitExp
%%{
    public String eval() {
        return lit.toString();
    }
%%}
```

In this example, `lit` is the only field in the `LitExp` class: it has type `Token`, and evaluating `lit.toString()` returns the string value of the token as it appears in the program source.

2.7 Putting it all together

The three sections of a PLCC specification – lexical, grammar, and semantics – appear in one file, with the sections separated by a line with a single '`%`' sign. Comments can appear in the PLCC lexical and grammar specifications starting with a `#` and continuing to the end of the line.

Here is a complete specification example. The evaluation semantics of a numeric literal is the literal itself, and the evaluation semantics of a variable symbol is the uppercase version of the symbol.

```
# a simple language with numeric literals
#   and variable symbols
# lexical specification
skip WHITESPACE '\s+'
skip COMMENT '%.*'
LIT '\d+'
VAR '[a-zA-Z]\w*'
%
# grammar rules
<program>    ::= <exp>
<exp>:LitExp ::= <LIT>
<exp>:VarExp ::= <VAR>
%
# semantics
Program
%%{
    public String toString() {
        return exp.eval();
    }
%%}

Exp
%%{
    public abstract String eval();
%%}

LitExp
%%{
    // return the literal string
    public String eval() {
        return lit.toString();
    }
%%}

VarExp
%%{
    // return the symbol in uppercase
    public String eval() {
        return var.toString().toUpperCase();
    }
%%}
```

2.8 Read-eval-print

PLCC automatically generates a read-eval-print `Rep` class whose `main` method repeatedly prints a prompt, creates an instance of the `Scan` class from `System.in`, and parses and prints the grammar start symbol class. A sample interaction running the Java `Rep` program using the above language specification looks as follows:

```
--> 42
42
--> xyZzY % should print XYZZY
XYZZY
```

3. PLCC ARCHITECTURE

I chose Perl to write PLCC because Perl has good string handling and pattern matching capabilities. The output of PLCC is a set of Java programs, all of which are text files and are generated in a subdirectory of the current directory named `Java`.

PLCC reads the token specifications and creates a file named `Token.java` that contains `enum` entries for both skip

tokens and normal tokens. PLCC generates these entries from the regular expressions given in the token specifications, turning them into Java `String`s with appropriate escaping. PLCC uses a template file called `Token.pattern` as the basis for filling in the appropriate token definitions drawn from the specification file to create the `Token.java` file. The end of the token specification is a line with a single `%`.

PLCC then reads the grammar rules and creates entries for each class, noting which classes must be abstract (because the nonterminal appears more than once on the left-hand-side of the grammar rules), and keeping track of the corresponding right-hand-sides. It checks the grammar for being LL(1) and reports an error if not. Each repetition grammar rule (with `**=`) is turned into multiple rules that use recursion instead of repetition, but only for the purpose of checking for LL(1).

Once the grammar is determined to be LL(1), PLCC generates class stubs for each of the grammar rules, as well as a `Rep.java` file to implement the read-eval-print loop. The class stubs include generated code for the `parse` methods specific to the grammar rules.

The `Rep.java` file is built from a template that only needs to have the name of the start symbol class filled in. The end of the grammar section is a line with a single `%`.

PLCC then reads the semantics specification entries, each of which starts with a class name followed by lines sandwiched between lines containing `%%{` and `%%}`. If the class name is one of the stubbed classes, the given lines are inserted into the stub class verbatim. If the class name is not one of the stub classes, PLCC creates a new class containing the given lines.

The `Scan.java` program is part of the PLCC standard code library and is automatically included in the generated code directory. Using the generated `Token` class, the `Scan` class does all of the dirty work of reading lines of the input stream (a `BufferedReader`), skipping over input that matches the `skip` specifications, and returning the next token.

All of the Java source files created by PLCC are deposited into a subdirectory named `Java`. Once these Java files are generated and compiled (errors in creating the semantic routines can be uncovered here), the `Rep` program can be run to test the resulting language implementation.

In order to make it easier to deal with the separate parts of the semantics of a PLCC specification, an `include` directive can be used in the semantics specification section, giving the name of the file to include in the specification. The contents of this file then become part of the PLCC specification input. This is useful, for example, to separate code that implements semantics specific to grammar classes from code that is used to implement auxiliary classes used in semantic actions. For particularly complex languages, it may be useful to have several of these files.

4. COMPARISON WITH OTHER PARSER GENERATORS

A Wikipedia comparison of parser generators[7] for deterministic context-free languages lists about 90 entries. Parsers for non-LL(1) languages are typically table-driven or backtracking, and the code generation for such languages is much more difficult for students to read and understand than the code for simple LL(1) predictive parsers (as generated by PLCC) based on recursive descent. Of the entries in this list, 12 of them clearly target languages that are LL(1) or that generate recursive descent parsers. Of these, only Coco/R[5] targets Java (the latest version also targets C# and C++).

However, Coco/R mixes syntax and semantics: semantic actions are specified in-line with the grammar rules, so Coco/R does not satisfy my goal of clearly separating syntax and semantics. Furthermore, the format of specifying Coco/R grammar rules is more complex than the simple BNF-style used in PLCC. Getting started using Coco/R is more difficult and time-consuming than learning how to use PLCC. I conclude that PLCC meets my course-related goals in a way that no other parser-generator does.

5. COURSE CONSIDERATIONS

I have used PLCC in two offerings of a Programming Languages course at my institution, and I plan to continue its use when I offer this course in the future. I have also used PLCC as the compiler-compiler for an offering of a Compilers class.

I have observed that my students learn how to use PLCC quickly, more so than when I was using `sllgen` and needed to cover the elements of `Scheme` from scratch. I have been able to cover more and richer language examples once I started using PLCC. Students coming into my Programming Languages class have experience with Java, but they generally do not have a good understanding of abstract classes: using PLCC gives them the opportunity to become familiar with abstract classes and how to use them.

6. PLCC AVAILABILITY

The entire PLCC toolkit consists of the `plcc` Perl program (1320 lines) and a set of template files used to to generate the scanner and read-eval-print loop (a total of 371 lines). These can all be downloaded from `http://cs.potsdam.edu/PLCC`.

7. REFERENCES

[1] C. Fischer and R. LeBlanc Jr. *Crafting a Compiler with C*. Benjamin/Cummings, Redwood City, CA, 1991.

[2] T. Fossum. Classes as first-class objects in an environment-passing interpreter. In *Proceedings of the Tenth Annual Conference on Innovation and Technology in Computer Science Education* (Lisbon,Portugal), pages 261-265. ACM, 2005.

[3] J. Friedl. *Mastering Regular Expressions*. O'Reilly Media, Sebastapol, CA, 2006.

[4] D. Friedman, M. Wand, and C. Haynes. *Essentials of Programming Languages (2nd ed)*. The MIT Press, Cambridge, Massachusetts, 2001.

[5] H. Mössenböck. A generator for production quality compilers. In *Springer Verlag Lecture Notes in Computer Science*, 477:42-55, 1990.

[6] Pascal: ISO Standard 7185. 1990. Retrieved December 2, 2013 from http://pascal-central.com/docs/iso7185.pdf.

[7] Wikipedia.Org. 2013. Comparison of Parser Generators. Retrieved September 5, 2013 from http://en.wikipedia.org/wiki/.