Piraha: A Simplified Grammar Parser for Component Little Languages

Steven R. Brandt [1], Gabrielle Allen [2]

Center for Computation and Technology
Louisiana State University
Baton Rouge, LA 70816, USA
{[1]sbrandt, [2]gallen}@cct.lsu.edu

***Abstract*—Software codes in scientific computing often implement their own little languages for expressing configuration data, interface definitions, and runtime parameters. Such languages are of particular importance for component-based frameworks. These languages can initially be somewhat ad-hoc and then expand organically.**

We describe how parsing expression grammars (PEG) can provide the capabilities for scientific application developers to easily construct appropriate parsers which will enable improved and more robust little languages.

We show how a little language parser could be used with the Cactus Configuration Language in the Cactus Framework.

I. Introduction

Little languages (defined as domain-specific languages which lack many features found in general purpose languages, see [1]) are common in the high performance community. They are used for a variety of purposes including configuration files and interface definition languages (Cactus, Babel, SIDL, and OnRamp [2]) as well as quality of service enforcements [3]. For a larger listing of little and domain-specific languages see [4].

More commonly, a number of scientific codes produce their own unique output formats. This situation is, to a degree, unavoidable as the data produced by simulations are too numerous to completely classify.

Even if one settles on the use of standard data languages as a medium for output, configuration, and initialization (XML, JSON, INI, etc.), it may be that there is a need to parse data extracted from these formats, e.g. a mathematical expression (with correctly grouped sub expressions).

These tasks are easily handled by grammar engines such as bison or javacc, but the trade-off is increased software dependencies and (for the researcher who is only familiar with regular expressions from Perl or Python) a significant learning curve in making the transition to the more complex tool (there is a new set of syntax to deal with, and a new set of problems to understand such as shift reduce conflicts).

As a consequence, in a small subset cases, a set of regular expressions is sometimes used as a quick and dirty substitution for a formal grammar parser. Perhaps initially, the use of a regular expression is easier and better if the syntax is sufficiently trivial. If, for example, an input parameter file for a science executable simply contains a string of lines that match the format "name = value". In this case no special tool is required to parse the input. Eventually, if the scientific program is successful and gains a user base, these files may get larger and may become more complex and feature creep may begin: comments, quoted strings, multi-line values, imports, and so on. Things may still work quite well for a while, but eventually the format may become unwieldy. At this point, time must be taken to refactor using a grammar parser such as yacc, or perhaps the developers must move to a more standard framework (XML, INI, etc.). Both of these options may be unattractive.

A Parsing Expression Grammar (PEG) [5] has the potential to answer this need. A PEG an analytic formal grammar that represents a recursive descent parser and resembles a set of regular expressions. Parsing expression grammars have the potential to be extremely simple since they do not require a separate step for tokenization; both the tokenization and the grammar rules are described by the same regex-like syntax.

The Piraha PEG engine implementation described here is designed to look and work in a manner similar to a regular expression API, but to provide the full power of a grammar parser. PEGs make this simplicity possible because they do not require a separate tokenization step, and because they use prioritized choice (the familiar grouping operation of a regex) rather than unordered choice description commonly used by context free grammars.

In this paper we report on the use of a PEG framework named Piraha inside a component-based scientific code named Cactus. The framework is a work-in-progress experiment to see if a simplified parsing expression grammar tool can ease the development and maintenance of little languages within a non-computer science community.

Use of our Piraha PEG API makes it easy for a research group to incrementally add functionality to the structure of its little languages, and to do it without a major refactoring or going through the learning curve required to master the use of a more formal grammar tool.

In Section II we described related work in PEG parsing packages and describe the differences between these and our new implementation. In Section III we describe PEGs in more details and use examples to illustrate the properties of our parser. In Section IV we describe a prototype imple-

mentation of a micro language using PEG for the Cactus Framework.

II. PEG Parsing Packages: Related Work

There are a number of Parsing Expression Grammars available, and all provide a number of benefits. Many of these are so-called packrat parsers which use memoization to parse any grammar in linear time. That is not our concern in this work, as parsing the small grammars we are considering rarely take significant time.

The Rats! parser [6] is a packrat parser designed to be easily extensible. It is implemented in Java and systematically outperforms many non-PEG parsers. The Rats! parser uses its own special grammar syntax file.

The pyPEG package [7] provides a simple framework in which Python functions are the rule definitions and their return values describe a token stream. Python regexes can be used to add elements to the token stream.

Boost Spirit [8] implements a fully type-checked parser that is constructed using the operator overloading mechanism of C++. It thus provides a little language within C++ to do the parsing.

Pyparsing [9] is a package which targets Python users. Pyparsing uses Python language objects to construct a PEG parser. Like pyPEG, it works directly on Python code and does not use a special syntax file.

The Piraha PEG parser we provide does not require a grammar file with a special syntax and does not make use of any special language features. It simply takes a sequence of named expressions (each of which are sequences of pattern elements that are familiar from other standard regular expression packages) and compiles them into a grammar. This framework minimizes the learning curve in moving from regular expressions to grammars and forms the basis of an API suitable for use in any programming language (currently, we have only implemented it for Java, but plan to make it available in Python and C++).

Note that while we do not require a specially formatted grammar file, for grammars of significant complexity it is useful to collect the patterns into a file that allows white space in the pattern, similar to the extended legibility flag (the "x" flag) compilation option familiar from Perl. This allows comments to be inserted (using the hash symbol) and maps white space to a pattern named "skipper" which the user can define.

III. Basic Syntax for the Piraha Parser

The notable differences between PEGs and regular expressions are that groups are always *independent non-capturing groups*, quantifiers are always *possessive*, and that patterns are named. Because patterns are named, they can be called recursively.

The pattern syntax we use for Piraha is similar to that used in common regular expression engines, and a summary of pattern elements is shown in Table V. Piraha is deliberately simpler than other engines, and the interface is made very similar to those used by Java's standard regular expression as this intends to be a tool for users who are not computer scientists.

Based on this minimal scaffolding, a parser for a mathematical expression can be constructed using just a few lines of code. The expression understands order of operations, grouping of parenthesis, and use of "**" as an exponentiation operator.

```
Grammar math = new Grammar();
math.compile("num",
    "([0-9]+\\.[0-9]+|[0-9]+\\.|"+
    "\\.[0-9]+|[0-9]+)"+
    "([eEdD](\\+|-|)[0-9]+|)"); // exponent

// The basic operators
math.compile("addop","\\+|-");
math.compile("mulop","\\*|/");
math.compile("powop","\\*\\*");

math.compile("neg","-");

// Note: left recursion is not supported
math.compile("expr",
    "{mulexp}{addop}{expr}|"+
    "{mulexp}|\\({-expr}\\)");
math.compile("mulexp",
    "{powexp}{mulop}{mulexp}|"+
    "{powexp}|\\({-expr}\\)");
math.compile("powexp",
    "{num}{powop}{powexp}|{neg}{num}|"+
    "{num}|\\({-expr}\\)");

Matcher m = math.matcher(
    "expr", "3.0*2.0+1.0e1*(2+4)");
if(m.matches()) {
    // do something with result
}
```

This will produce an easily handled tree of data that can be traversed as in the example above.

In order to facilitate use in component based systems, grammars can be imported one inside the other. This allows one syntax to be embedded inside another, similar to the way javascript is embedded in script tags, or PHP switches between code and html elements. Here is a simple example that imports the math grammar above.

```
Grammar g = new Grammar();
g.importGrammar("math", Calc.makeMath());
g.compile("vector",
    "{math:expr}(,{math:expr})*");
Matcher m =
    g.matcher("vector","1+2,(8+3)*9-4,4,9+7");
if(m.matches()) {
    System.out.println(
        "match: "+m.substring());
    // prints: match: 1+2,(8+3)*9-4,4,9+7
}
```

IV. Example Application: Cactus Framework

In this Section we illustrate how PEG could be used with an existing component framework. The Cactus framework [10] is an open source component framework with a large number of developers and users in the fields of numerical relativity, quantum gravity, computational fluid

dynamics and coastal modeling. For example, in the field of numerical relativity, over 140 Cactus components have recently been released as the *Einstein Toolkit* [11] which provide a community resource for researchers investigating systems such as black holes and neutron stars which are governed by Einstein's theory of general relativity.

Cactus components (called *thorns*) interact with the framework (called the *flesh*) via a set of configuration (CCL) files provided by each thorn. These files describe the interface of the thorn via its variables, parameters, functions and capabilities. The configuration files are parsed by Cactus at compile time to generate source code that is then compiled into the final Cactus executable. Additional tools have been developed that also parse the CCL files to provide information needed to more easily assemble or debug codes.

Here we investigate the use of PEG with the Cactus `schedule.ccl` file which describes the functions provided by the thorn including which of the standard Cactus time bins the function should be run in and when and how variable storage should be allocated. The syntax for the entries in the `schedule.ccl` file is shown in Fig. IV.

For grammars of this complexity, it is desirable to move the definitions into a file (which uses a .peg suffix by convention). Presence of white space between pattern elements results in the insertion of a pattern named "skipper". The skipper pattern can be used to handle arbitrary amounts of white space and comments between pattern elements.

For this reason, the space character needs to be escaped in order to be matched (unless it is inside square brackets).

This particular file is interesting because of the fairly general "if" clause it supports. It checks whether parameters are set or not, allows for logical and, or, and not, as well as parenthetical grouping.

The parser generated by this set of expressions is more compact, robust, and flexible than the current Perl-based parser in Cactus.

Notice the use of the pattern "`\\[\r\n]`" inside the skipper. The Perl-based parser required the arguments of the scheduler to all be placed on a single line, and the escaped carriage return (a convention borrowed from the C-preprocessor) was invoked to break up long lines. Because the PEG is able to more cleanly express and parse the required sequence of expressions, this syntactic artifact is no longer needed (but can still be supported).

A more complex example of a grammar is the param.ccl file used by Cactus (see Fig. IV). The grammar is over one hundred thirty lines long. For this reason, we only show a subset of it.

In the process of constructing and testing the grammar above, we discovered a number of irregularities in the Cactus component descriptions that the perl-based parsers failed to detect (i.e. in some places the documented comment syntax was violated but somehow passed validation, in another

```
skipper=\b({w}|{-ccom}|\#[^\n]*|\\[\r\n])*

w = [ \t\n\r\b]
any = [^]
name = (?i:[a-z_][a-z0-9_\-]*\b)
vname = {name}( :: {name})*( \[ {num} \]|)
quote = "(\\{any}|[^"])*"
ccom = /\*((?!\*/){any})*\*/
num = [+\-]?[0-9]+
string = {name}|{quote}
term = {num}|{name}
par = \b(as|at|in|while|if|before|after|while)\b
pararg = ({vname}|\( {vname}( ,? {vname} )+ \))

boolterm = (?i:\! {boolexpr} | \( {boolexpr} \)
         | CCTK_Equals \( {string} , {string} \)
         | CCTK_IsThornActive \( {string} \)
         | \*? {name} )

boolexpr = {boolterm} ((&&|\|\|) {boolexpr} )+ |
         | {term} (>=|<=|==|!=|<|>) {term}
         | {boolterm}

schedule = (?i:
 schedule (group|) {name} ({par} {pararg} )* \{
  ( storage : {vname}( , {vname}|[ \t]{vname})*
  | lang(uage|) : {name}
  | sync : {vname}( , {vname}|[ \t]{vname})*
  | options : {vname}( , {vname}|[ \t]{vname})*
  | triggers : {vname}( , {vname}|[ \t]{vname})*
  | trigger : {vname}( , {vname}|[ \t]{vname})*
  )*
 \} {quote}
)
if = (?i:
    if \( {boolexpr} \) {block}
       (else {if}|else {block}|)
  )
storage = (?i:storage: {vname}( ,
  {vname}|([ \t]|\\\r?\n)+{vname})* )
block = \{ ({statement} )* \} | {statement}

statement = ({schedule} |{if} |{storage} )
sched = {-skipper}{statement}*$
```

Figure 1. Basic parser for a Cactus schedule.ccl file

a required parameter was missing). We also discovered features that were in place but not widely known.

The reason for the length of the param.ccl grammar is that it needs to parse a number of different kinds of parameters (integer, real, string, keyword, and boolean), each repeated with minor variations on its allowed syntax. This example has been trimmed to only show the integer parameter for the sake of brevity.

One of the advantages of the PEG framework is that it would allow other types of parameters to be easily added, say a complex number or vector.

Of interest here is the parsing of the range. There is a lower and an upper range that can be specified in setting the allowed values for each parameter, and the mathematical use of [and (to specify whether the value given is inclusive or exclusive.

V. Conclusion

We have demonstrated a simple and flexible framework for parsing little languages that should prove useful in component development work, enabling more powerful and

```
uses = (?i:uses|);

skipper = \b([\ \t\n\r\b]|\#[^\n]*|\\[\r\n])*

any = [^]
name = (?i:[a-z_][a-z0-9_]*)
accname = {-name}(::{-name})*
steerable = (?i:never|always|recover)
accumexpr = \( ([^()]+|{accumexpr}) \)

access = (?i: global : | restricted : |
    private : | shares :([\ \t]*{name}|) )
quote = "(\\{any}|[^"])*"

num = [+\-]?[0-9]+
intbound = \* | {num} |
intrange = [\[\(]?{intbound} :(?!:)
    {intbound}[\]\)]? | {intbound}

intguts = (?i:
    (CCTK_|)INT {name}(\[{num}\]|)
    ({quote}|)
    (as {name} |)
    (steerable = {steerable}|)
    (accumulator = {accumexpr} |)
    (accumulator-base = {accname} |)
  )

intpar = (?i:
    extends {intguts}
    ( \{
      ( {intrange} (:: {quote}|) )*
    \} |)|
    {uses} {intguts}
    ( \{
      ( {intrange} (:: {quote}|) )*
    \} {num}|)
  )

...

pars = ^ ({+access}|{+intpar}|{+realpar}|
    {+keywordpar}|{+stringpar}|{+boolpar})* $
```

Figure 2. A subset of the grammar required to parse the param.ccl file used by Cactus.

correct parsing of configuration information or runtime parameters, and have demonstrated its potential value for the Cactus framework.

To make it more attractive in the high performance world we plan to support implementations in both C++ and Python. It is expected that these API's can be generated automatically by using the PEG framework itself to parse the core Java code.

Table I
SUMMARY OF PIRAHA PATTERN SYNTAX

.	Any character but but \n
[a-gx]	Match an entity. Entities are ranges of character or literals. This example matches characters in the range a through g and x.
[^a-gx]	Match an entity, but only if the character does not fall within the ranges or literals specified.
[^]	Match any character.
x{n,m}	The pattern x occurs a minimum of n times and a maximum of m times
x{n,}	The pattern x occurs a minimum of n times
x{,m}	The pattern x occurs a maximum of m times
x*	The pattern x occurs zero or more times
x+	The pattern x occurs one or more times
x?	The pattern x occurs zero or one times
\1-\9	Match the nth backreference within this rule
{name}	Match the pattern named "name" and capture matching text in a backreference
{-name}	Match the pattern named "name" and do capture
(x|y|z)	Specifies a grouping of patterns and alternative possible matches
(?=x)	A zero-width lookahead assertion
(?!x)	A zero-width negative lookahead assertion
\b	A word boundary
$	Matches the end of input
^	Matches the start of input
(?i:x)	Ignore case when matching x
(?-i:x)	Don't ignore case when matching x

REFERENCES

[1] M. Mernik, J. Heering, and A. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys (CSUR)*, vol. 37, no. 4, p. 344, 2005.

[2] G. Hulette, M. Sottile, B. Allan, and R. Armstrong, "OnRamp to CCA: Annotation-driven static analysis and code generation," 2009.

[3] L. Li, T. Dahlgren, L. McInnes, and B. Norris, "Interface contract enforcement for improvement of computational quality of service (CQoS) for scientific components," in *Proceedings of the 2009 Workshop on Component-Based High Performance Computing*. ACM, 2009, pp. 1–5.

[4] A. Van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *ACM Sigplan Notices*, vol. 35, no. 6, p. 36, 2000.

[5] B. Ford, "Parsing expression grammars: a recognition-based syntactic foundation," *ACM SIGPLAN Notices*, vol. 39, no. 1, p. 122, 2004.

[6] R. Grimm, "Practical packrat parsing," *New York University Technical Report, Dept. of Computer Science, TR2004-854*, 2004.

[7] pyPEG, http://fdik.org/pyPEG/.

[8] J. de Guzman *et al.*, "The Boost Spirit Library."

[9] P. McGuire, "Pyparsing: a general parsing module for Python," *URL http://pyparsing. wikispaces. com*.

[10] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf, "The cactus framework and toolkit: Design and applications," *High Performance Computing for Computational Science VECPAR 2002*, pp. 15–36, 2003.

[11] E. Schnetter, "Multi-physics coupling of Einstein and hydrodynamics evolution: a case study of the Einstein toolkit," in *Proceedings of the 2008 compFrame/HPC-GECO workshop on Component based high performance*. ACM, 2008, p. 4.