Tree Building Parsing Expression Grammars

James Pike

April 12th, 2010

Abstract

Parsing Expression Grammars (PEGs) provide an alternate recognition-based formal foundation for describing machine oriented syntax[**?**]. This paper suggests a notation based on that of PEGs called Tree Building Parsing Expression Grammars (TBPEGs) that allows both the syntax of the grammar and an appropriate tree structure to store the result of parsing source strings of this grammar to be specified simultaneously. The next section will discuss chilon::parser, a C++0x library that allows TBPEGs to be specified with templates. chilon::parser is capable of processing an input grammar to create a suitable Abstract Syntax Tree (AST) type at compile time using meta-programming, and of populating this AST at run-time from input source texts. This paper suggests that TBPEG based parser generators offer a powerful and concise alternative to existing parser generator state of the art.

1   Introduction

This paper will concentrate on the use of TBPEGs to aid the process of writing a programming language compiler. It is expected that this is one of the more complicated use cases of TBPEGs, although it is anticipated this work could prove useful in other areas also.

When writing a compiler, the author must first create an abstract syntax tree (AST) using data structure facilities provided by the compiler language, to store the parsed source code in a format suitable for further processing. Machine based languages are unambiguous with a regular structure by design and thus the structure of the AST is often very similar to the structure of the grammar itself. This suggests that a lot of work the programmer puts into creating the AST could be removed and handled by a parser generator based on the grammar description. The act of translating parsed text to the AST nodes the programmer has already created is generally handled with the use of a parser generator such as ANTLR[**?**] that reads a Context

Free Grammar based input language. Semantic actions can be attached to rules that are responsible for taking strings obtained by parsing rules specified in the grammar and copying this data into AST nodes that must also be created by the same grammar rule or a parent rule. It follows that if both the type of the AST and the method of parsing text can be derived from the same grammar, then the process of populating the AST can be automated as well. The boost::spirit[**?**] PEG based parsing framework is capable of determining apropriate C++ storage containers based on rules in the grammar.

This paper will show how a TBPEG based library can be used to remove the burden of AST construction and population from the programmer. The first section of this paper will discusses the differences between TBPEG and PEG parsing behaviour. The next section will will show the rules TBPEGs use to create storage types for expressions. The final section will discuss the implementation and usage of chilon::parser, a C++0x library which implements the TBPEG notation directly in C++0x using template metaprogramming. This section includes an example to show how chilon::parser can be used to create a program that can scan TBPEG grammars and output chilon::parser source code. The result of using this program to bootstrap an equivalent program using its own grammar is discussed.

2   Differences between TBPEG and PEG parsers

TBPEGs behave identical to Parsing Expression Grammars[**?**] except for the following deviations.

2.1   The "!" operator does not create a not-predicate for character matching expressions

In PEGs the not-predicate is specified with the "!" operator and fails when the sub-expression matches otherwise succeeds and never consumes. In TBPEGs if the sub-expression of a "!" operator matches a character then it consumes that character.

A PEG that uses a not-predicate:

```
Tag <- "<" (! ">" .)+ ">"
```

The equivalent TBPEG:

```
Tag <- "<" (! ">")+ ">"
```

A "!" operator must be prefixed with "&" to make it a not-predicate regardless of whether it matches a single character or not:

```
NonPVariable <- &!"p" Variable
```

2.2 Spacing

"Spacing?" is allowed to match between elements in a sequence. The operator ∧ is used between elements where whitespace is not permitted to match. ∧ has higher precedence than the sequence operator. Multiple expressions joined with the ∧ operator are called a "joined sequence". The operator ∼ is provided to separate items in a sequence where Spacing is mandatory and has the same precedence as ∧. A joined sequence where every sub-expression is a lexeme, character or string matching expression is a "lexeme". The rule "Spacing" may not refer to itself, therefore it may not contain sequences, joined sequences are permitted.

This PEG:

```
Spacing             <- \s
Expression          <- [0-9]+
Identifier          <- [a-zA-Z_][a-zA-Z0-9_]
Assignment          <- "=" Spacing* Expression
VariableDeclaration <- "var" Spacing+ Identifier (Spacing* Assignment)?
```

Is equivalent to this TBPEG:

```
Spacing             <- \s+
Expression          <- [0-9]+
Identifier          <- [a-zA-Z_] ∧ [a-zA-Z0-9_]
Assignment          <- "=" Expression
VariableDeclaration <- "var" ∼ Identifier Assignment?
```

Spacing is optional between elements in a "+" or "*" repetition unless the sub-expression is a character match. "∧+" and "∧*" are equivalent to "+" and "*" in PEG notation.

This PEG:

```
Spacing    <- \s
Identifier <- [a-zA-Z]+
Arguments  <- Identifier (Spacing+ Identifier)+
String     <- ["] (\\. / ! ["] .)* ["]
```

Is equivalent to this TBPEG:

```
Spacing    <- \s+
Identifier <- [a-zA-Z]+
Arguments  <- Identifier+
String     <- ["] (\\ ∧ . / ! ["] )∧* ["]
```

2.3   Join expressions

The PEG:

```
Arguments <- Identifier (Spacing? "," Spacing? Identifier)*
```

Is identical to the following TBPEG:

```
Arguments <- Identifier % ","
```

To disallow Spacing to match around join expressions ∧% can be used.

```
Addition <- Identifier ∧% (\s* "+" Spacing)
```

3   Storage types of TBPEG expressions

Every expression defined in TBPEG notation has an associated storage type. In cases where storage is not appropriate for an expression the storage type is "void". An expression with a void storage type is a "constant expression", and an expression with a non-void storage type is a "storing expression".

3.1   Constant TBPEG expressions

Expressions that always match the same data are constant.

```
VarDeclarationPrefix <- "var"
LongSuffix           <- "l"
```

Enclose a character in brackets to force it to have character storage type:

```
StoresP <- [p]
```

The expression "Spacing" is constant, as are predicates, $\backslash r$, $\backslash n$, $\backslash t$, $\backslash n$, $\backslash s$ and the empty string ε.

Repetitions of constant expressions are constant.

```
EmptyLines <- EndOfLine+
```

Sequences and joined sequences in which all sub-expressions are constant are also constant.

```
Spacing <- \s+
```

3.2 Storing character matches

TBPEG Storing expressions that match a single character have a storage type of "character":

```
AnyThing  <- .
LowerCase <- [a-z]
```

A "!" operator applied to a character matching sub-expression is always a storing expression of type character.

```
StoresCharacter <- ! "p"
StoresNull      <- &! "p"
```

3.3 Storing repetitions of character matches

A repetition of a character storing expression stores a "string". When using the "*" operator this string may be empty:

```
Identifier        <- [a-zA-Z]+
OptionalIdentifer <- [a-zA-Z]*
```

3.4 Storing joined sequences with strings

Joined sequences of expressions that store "character" or "string" store "string":

```
Identifier <- [a-zA-Z_] ∧ [a-zA-Z0-9_]+
```

3.5 Storing sequences with tuples

Sequences with only a single storing sub-expression store the same type as this sub-expression.

```
VarDefinition <- "var" Identifier
```

Sequences containing multiple storing sub-expressions have storage type "tuple". The number of elements the tuple stores is equal to the number of storing expressions in the sequence and the type of each tuple element is equivalent to the type of the corresponding sub-expression.

A PEG expression that stores tuple(string, string):

```
FirstnameAndSurname <- [a-zA-Z]+ [a-zA-Z]+
```

A PEG expression that stores tuple(character, string):

```
InitialAndSurname <- [A-Z] ∧ "." [a-zA-Z]+
```

If any sub-expressions of the sequence also have a storage type of tuple, then this tuple is collapsed into the parent tuple. In this case the number of elements in a tuple is equal to the number of storing non-tuple elements added to the number of elements in all tuple sub-expressions.

In this PEG VariableDefinition stores tuple(string, string) and TypedVariableDefinition stores tuple(string, string, string):

```
VariableDefinition      <- Identifier "=" [0-9]+
TypedVariableDefinition <- Identifier VariableDefinition
```

Joined sequences follow the same storage rules as for sequences, except adjacent string/character matches collapse to form longer string storage types.

In this TBPEG FunctionDeclaration stores tuple(string, string):

```
FunctionDeclaration <- Identifier ∧ ("()"  "->") ∧ Identifier
```

3.6 Storing ordered choice

If all of the sub-expressions of an ordered choice store the same type, then this is also the storage type of the ordered choice.

```
IdentifierBegin <- [a-z] / [A-Z]
```

A variant type is modeled after a boost::variant[**?**]. A variant(U) has a set of types U and may store any one of the types in U. It also stores state to indicate which of the types in U is currently stored.

An ordered choice where each sub-expression stores a different type has a storage type of variant(T). T is the set of unique storage types from all sub-expressions of the ordered choice. Storage type is determined based on the storage type of the sub-expression and may be void with certain exceptions. Sub-expressions that are constant character matches have storage type "character" rather than void. All other sub-expressions with void storage type have storage type "string" except for predicates and the empty string ε.

In this TBPEG VariablePrefix stores variant(void, string) and Expression stores variant(tuple(string, string), string):

```
Identifier     <- [a-zA-Z]+
VariablePrefix <- "var" / "val" / Identifier
Expression     <- Identifier / (Identifier "," Identifier)
```

If any sub-expressions of an ordered choice also have a storage type of variant, then this variant is collapsed into the parent variant.

In this TBPEG FunctionPrefix stores variant(void, character, string):

```
Identifier     <- [a-zA-Z]+
VariablePrefix <- "var" / "val" / Identifier
FunctionPrefix <- "f" / VariablePrefix
```

3.7   Storing "optional"

Optional expressions are stored using variant(void, S) where S is the storage type of the sub-expression. In this TBPEG Prefix stores variant(void, string):

```
Prefix <- ([a-zA-Z]+)?
```

3.8   Storing repetitions of non-character matches

Repetitions that use $\wedge*$ and $\wedge+$ where the sub-expression stores a character store "string".

```
String <- ["] (\\ . / ! ["] )∧* ["]
```

Expressions which are repetitions of a sub-expression store list(S), where S is the storage type of the sub-expression. In this TBPEG Arguments stores list(string) and TypedArguments stores list(tuple(string, string)):

```
Identifier     <- [a-zA-Z]+
Arguments      <- Identifier+
TypedArguments <- (Identifier Identifier)+
```

3.9   Storing join expressions

The storage type of a join expression is equal to tuple(list(S), list(T)) where S is the storage type of the expression on the left hand side of the % operator and T is the storage type of the expression on the right hand side. If T is void then list(S) is the storage type of the join expression.

In this TBPEG Arguments stores list(string) and Binaries stores tuple(list(string), list(char)):

```
Identifier <- [a-zA-Z]+
Arguments  <- Identifier % ","
Binaries   <- Identifier % [+-]
```

3.10   Node rules for solving ambiguity

In addition to rules defined with <-, a "node rule" can be defined using <=. Node rules are passed in the same way as non-node rules but the storage type of a node rule can never be broken down. For example a node rule that stores a tuple will be stored as a single element inside the tuple storing super-expression.

Node rules introduce a new type into the type system and can be used to resolve ambiguity. A node rule named P will have storage type S(P), and a TBPEG parser generator would make S(P) a class or structure named P with an appropriate attribute created to store the matched data in. In the following TBPEG, Expression stores variant(S(Identifier), string). If Identifier was not a node rule then it would be impossible to tell whether a string or Identifier reference was stored in an AST node created by the Expression rule.

```
Identifier   <= [a-zA-Z]+
String       <- \"  (! \")+ \"
Expression   <- String / Identifier
```

Node rules can also be used to resolve ambiguity when determining the storage type of self-referential rules. In the following TBPEG the AST type of Expression would contain infinitely nested lists if it was not a node rule:

```
Number     <- [0-9]+
Primary    <- "(" Expression ")" / Number
Expression <= Primary %+ "+"
```

3.11   Tree rules for eliminating redundant AST nodes

Consider AST structure for a simple mathematics grammar.

```
Spacing    <- \s+
Number     <- [0-9]+
Term       <- Number / "(" Expression ")"
Product    <= Term %+ "*"
Addition   <= Product %+ "+"
Expression <= Addition
Grammar    <- Expression+
```

The storage type a parser generator might create for this TBPEG in C++ might look like this:

```
class Expression;
class Product {
    vector< variant<string, Expression> > value_;
}
class Addition {
    vector<Product> value_;
}
class Expression {
    vector<Addition> value_;
}
typedef vector<Expression> Grammar;
```

The text:

```
4 7
```

Would be stored in the the following AST which contains many redundant nodes:

```
[ Expression(Addition[Product[4]]),
  Expression(Addition[Product[7]]) ]
```

The storage type of this grammar can be simplified using a `|%` rule. A rule of the form `P <= Q |% R` has storage type `variant(S(P), storage type of Q)` and at least one `Q` must match. In this case `S(P)` will have an attribute capable of storing `vector(storage type of Q)`. If a single `Q` is matched then the latter storage type is used, but if two or more `Q` match joined by `R` then the storage type of `Q` is used. Using this and relying on recursive expansion of variant storage types the storage type of the mathematics grammar can be simplified:

```
Spacing    <- \s+
Number     <- [0-9]+
Term       <- Number / "(" Expression ")"
Product    <= Term |%+ "*"
Addition   <= Product |%+ "+"
Expression <= Addition
Grammar    <- Expression+
```

In this case the storage types contain more variants that enable redundant tree nodes to be skipped:

```
class Expression;
class Product {
    vector< variant<string, Expression> > value_;
}
class Addition {
    vector< variant<Product, string, Expression> > value_;
}
class Expression {
    variant<Addition, Product, string, Expression> value_;
}
typedef vector<Expression> Grammar;
```

The same text:

```
4 7
```

Would be stored in the neater AST:

```
[ Expression(4), Expression(7) ]
```

There are more tree rules:

```
P <- Q|+
```

This stores `variant(P, storage type of Q)`, depending on whether one or more `Q` matches. At least one `Q` must match and `S(P)` will contain an attribute capable of storing many `P`.

```
P <- Q R|?
```

This stores `variant(P, storage type of Q)`, depending on whether `R` matches or not. `S(P)` will contain an attribute capable of storing both `Q` and `R`.

`|^%` and `|^+` are also provided which match and act the same as their equivalent non ∧ rules but which respect the whitespace rules of their non-tree equivalents.

4   chilon::parser, a C++0x implementation of TBPEGs

TODO

5   trek.js, a JavaScript implementation of TBPEGs

TODO

5.1   A chilon::parser TBPEG to chilon::parser translator

TODO

References

[1] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. ACM SIGPLAN Notices, 2004.

[2] TJ Parr, RW Quong. ANTLR: A predicated-LL (k) parser generator. Software-Practice and Experience, 1995.

[3] E Friedman, I Maman. Boost Variant. boost.cowic.de, 2002.

[4] J de Guzman. Boost Spirit Parser Generator Framework. http://spirit.sourceforge.net, 2010.