ANTLR Yggdrasil Manual

Loring Craymer

Draft

Introduction

Welcome to ANTLR Yggdrasil! ANTLR Yggdrasil is a variant/extension of ANTLR, a powerful language translation package developed by Terence Parr. ANTLR Yggdrasil differs from the baseline ANTLR 3 in terms of both capabilities and philosophy of language translation. With ANTLR Yggdrasil, you should never need to depend on the capabilities of the target language when developing a translator; instead, the needed capabilities are directly supported by Yggdrasil syntax.

Yggdrasil is about syntax trees, and it implements a concept that I call "Tree Attribute Grammars". Yggdrasil focuses on syntax trees, their decoration and construction. Thus the name "Yggdrasil": in Norse mythology, Yggdrasil is the World Tree that connects the nine realms of the Norse cosmos.

Philosophically, Yggdrasil is driven by the following:

1. The goal is rapid development of highly capable language processors.
2. Syntax trees are fundamental, and tree restructuring is the basic mechanism for language translation. The ^ and ! annotation developed by Parr for SORCERER is a powerful base for such restructuring, but the idea should be taken further. No restructuring code should occur in actions defined in the target language.
3. Users should not have to manually derive tree walker grammars. That is something that can be done automatically.
4. Translators be target language independent. It should take minimal effort to re-target a language translator to another implementation language.
5. Attributes should be first class syntactic elements of ANTLR. Attribute manipulation, including the retyping of tokens or syntax tree nodes and the replacement of token text, should be directly expressible in ANTLR/Yggdrasil syntax.
6. Code for attribute classes and other datatypes used in an Yggdrasil application should be generated by Yggdrasil whenever possible.
7. While target language actions are to be avoided during the process of translating language input, the translator may have to be integrated with code that processes the data once it is in that internal form. Not all translation applications need back-end processing, but integration with a back-end processor written in the target language should be painless.

Language Composition

ANTLR/Yggdrasil is composed of multiple domain specific languages. These are

1. An EBNF dialect for character recognition and token identification.
2. An EBNF dialect for parsing.
3. An EBNF dialect for tree walking.
4. Yggdrasil—a language for defining and manipulating attributes for constructing syntax trees and graphs.
5. String Template—a language for processing attributes into text.

Language composition is not new--lex/yacc probably provides the seminal example—but the use of composition as a pattern for development seems not to have been previously formalized. ANTLR Yggdrasil not only shows the power of language composition—as a result of composition, it cleanly supports very rapid development of language processors—but it also provides a framework for composing grammars.

Yggdrasil is a domain specific language for attribute definition and manipulation. It implements an attribute algebra that is conceptually simple, but capable of generating very complex data structures for back-end processing. It is implemented as an island grammar integrated with the EBNF dialects from ANTLR 2.

String Template is an output language. Despite a deceptively simple syntax, it provides a language for solving almost any text generation problem imaginable. At this point, String Template has a standalone grammar; this is likely to continue to be the case as part of the String Template language is "any sequence of characters except an unescaped '$' or '<'": these represent data not processed by the StringTemplate runtime library.

ANTLR also integrates "pluggable" support for target languages—Java, C#, C/C++, etc.--to gain access to processing capabilities beyond those supplied by the five DSLs.

Attributes in Yggdrasil

Attributes are characteristics of objects. For our purposes, attributes are either represented by data items or "fields" of the object or are computed from fields in the object. In Yggdrasil, attributes may be associated with 1.) grammars, 2.) tokens, and 3.) AST nodes. Attributes may be hierarchical: compound attributes may themselves have attributes.

Attributes are translated to target language objects, and Yggdrasil attributes are strongly typed. Yggdrasil has syntax to directly support manipulation of attributes; except for the ^ and ! rewrites, tree rewrites are accomplished by assigning tokens or AST nodes to grammar attributes and later referencing them to instantiate them as nodes in the generated syntax tree.

Conceptually, the grammar-level attributes comprise a blackboard which can be written to or read from. There is dynamic scoping support for rules; that is, the current value of a grammar-level attribute can be saved on entry to a grammar rule and set to another value—including a "blank" attribute of the same type.

Attribute Typing

One unusual feature of Yggdrasil is that attributes are strongly typed. That is, attribute data types are defined as in the example below:

```
@{
        native atomic int;
        native atomic String;
```

```
    Payload {
        int type;
        String text;
    }
}
```

Yggdrasil then knows that an attribute of type "Payload" has attributes "type" of type int and "text" of type String. The "native" keyword specifies that this type is provided by the target language; an atomic type is primitive (no code is generated for it). This example is contrived, in the sense that both int and String are built in primitives that are automatically translated by Yggdrasil into their target equivalent: in some possible targets, "int" would become an Integer object.

Yggdrasil built-in types consist of

1. atomic types
    1. String
    2. Integer
2. Compound types
    1. Payload (type, text)
    2. Carrier (down, right, type)
    3. Queue
    4. Table
    5. Template
    6. Library (StringTemplateGroup)
    7. Object (everything is an Object, but when referenced, it is used as a catchall base for compound type definition)

Additionally, user-defined atomic types are supported, and users can subclass any of the compound types. Attribute support for lexers is currently weak; it may be necessary to add character queues and possibly other character types.

Yggdrasil files have named headers, much like ANTLR 2. Following the file headers is an attribute declarations section that includes attribute type definitions; an "import <definitions file>" statement is also supported to allow reuse of attribute type definitions across grammars.

The Yggdrasil Attribute Algebra

Yggdrasil supports four operations on attributes. Attributes may be

1. Assigned.
2. Instantiated in the syntax tree.

3. Constructed from other attributes; this includes construction of syntax tree fragments.

Attributes may also be computed; however, there is no special syntax for computed attributes. The computation simply occurs when an attribute is assigned to or read from.

This simple set of operations is sufficient to provide complete rewrite functionality. The syntax for these operations can be mingled with the ^ and ! annotation taken from SORCERER and ANTLR 2 and is intended to be intuitive and easily used. [Cases where it is not should be reported to the author so that they can be fixed.]

Assignment occurs in two forms: with instantiation, and without. Examples:

```
a = B
```

Here, B is a token in the token stream to be recognized. It is then assigned to attribute a (a grammar-level attribute) and instantiated in the output tree. In this example, a can be either a Carrier or a Payload data type; more on that below.

```
a.b.@c = D
```

Again, D is a token to be recognized. It is assigned to the c "field" of the attribute occupying the b field of grammar-level attribute a. (In other words, it behaves just like the equivalent Java assignment statement.)

The only syntactic difference between the two forms of assignment is the way in which the assigned attribute is referred to. If prefixed with an @ sign, it is instantiated as well as assigned. Otherwise, it is assigned and not instantiated in the syntax tree unless that is done separately.

Yggdrasil is permissive about assigning Payloads or Carriers to fields. That is, assignments such as

```
@a.type = D
```

are allowed; this would be a fairly normal shorthand for "a.type to D.type; discard D". Runtime support, is the responsibility of the developer—unless target code is written to support such an assignment, the generated code is unlikely to compile.

One of the items that was never taken off of the ANTLR 2 "to do" list was subrule labeling. The idea was that a labeled subrule such as sub:(A B^ C)* had a restricted scope for the ^ operation. Yggdrasil achieves this via assignment of a subrule:

```
@sub = (A B^ C)*
```

assigns the tree generated from the subrule to the sub attribute. This is logically consistent: by assigning the subrule to an attribute, we are removing it from the output stream without side effects on other tokens matched within the rule. We could also do

```
sub = (A B^ C)*
```

to instantiate the subrule tree; this is exactly what the labeled subrule concept was intended to accomplish.

Instantiation has even simpler syntax. The attribute is simply referenced with the @ prefix:

```
@a
```

instantiates attribute a. A typical use of assignment and instantiation to rearrange a syntax tree is

```
@a = B
C D
@a
```

This results in (C D B) in the output AST.

An alternative view of assignment and instantiation is that attributes can either be used as labels (a = A) or as input/output variables: @a = A can be interpreted as "A is input to a", and @a can be interpreted as "A is output from a into the tree undergoing construction).

Construction of attributes is usually coupled with assignment, and attributes are constructed with an @(...) syntax. To construct a Payload in the output tree, for example, we might do

```
a = A
b = B
@( a.type b.text)
```

to construct a Payload that is of the same type as A, but has the text from B. The Payload would be of default type for the grammar.

Alternatively, we can couple construction with assignment to take advantage of attribute typing:

```
@a = A
@b = B
c = @(a.type, B)
```

If attribute c has been declared as a MessyPayload, this would result in a call to

```
new MessyPayload(A.type, B)          // Java or C++
```

for construction since anything assigned to c must be a MessyPayload.

Constructed attributes need not be Payloads. Any attribute type may be constructed and assigned. However, the developer will need to define the appropriate constructors in the target language as part of the definition of an attribute type.

Special features

Yggdrasil supports output rules:

```
rule @: ^( @a @b @c ) ;
```

produces an output tree, but does not recognize any input. While this is probably not strictly needed, it does provide support for factoring tree construction to improve readability.

Within a rule, the current tree is referenced by "@$". You can assign @$, kill it (@$!), or copy it (@$.copy) or reference fields in the current root or start node (this when the rule is constructing a sequence of nodes and does not have a root yet assigned).

Tree Construction

Tree construction in Yggdrasil follows the SORCERER and ANTLR 2 approach of using ^ and ! for structuring. I experimented with a tree construction syntax in ANTLR 2.8, but the attribute algebra of Yggdrasil makes anything beyond the ^ annotation unnecessary. To instantiate attribute a as the root of the current tree, simply reference @a^.

One feature that I have kept from ANTLR 2.8 is the idea of predicated tree construction. One of the objectives in designing tree structures is to remove semantic ambiguities, and so semantic differences are converted to syntactic differences in tree structure. Thus we might differentiate a keyword reference from a variable reference by constructing different trees

```
@a = A
@b = B
@c = C
@{     ?{ vars.contains(a) }  @a  @b  @c
       |
                @a^  @b  @c
}
```

to decide whether to construct A B C or #(A B C).

One of the difficulties in using semantic predicates in conjunction with backtracking is that semantic context might change dynamically. Yggdrasil "solves" this problem by distinguishing between the use of semantic predicates for recognition and for construction. During recognition, attributes are not altered. Instead, attribute modification occurs during tree construction at a time when there can be no backtracking and changes in semantic context are deterministic. So for languages with truly ugly ambiguities—like C++--the general translation approach is to capture semantic differences in structuring the syntax tree and defer recognition until the next tree walker pass[1].

Target language translation of the attribute algebra

In general, attribute references are translated into "get/set" method calls. In Java, at least, neither class nor attribute names are mangled; this may change for other languages. C "class" names will likely require mangling due to the lack of package/namespace support.

Code generation and adaptation

Yggdrasil can generate code for all attribute classes. In fact, this was one of the primary motives for having strongly typed attributes: users should not have to create support code by hand that can be created by machine. The generated code is fairly simple—a generated class contains fields, get/set methods, and skeletal query functions that always return "false".

1 Yggdrasil does not have a mechanism for detecting "improper" use of semantic predicates, and the developer is free to violate this principle. However, there is no guarantee that the generated translator will correctly recognize dynamic context.

Most generated classes will be "good enough", but many will need to be customized. Areas of customization include

1. Recoding of query functions to provide utility.
2. Addition of set<attribute>(CustomPayload) methods.
3. Addition of computed get/set methods where desirable.

Regeneration of attribute type classes is a normal part of development, but care should be taken to avoid overwriting customized code. I find diff3 very useful for this. Invoking (GNU) diff3 -m <myfile> <base file> <new file> > tempfile; mv tmp <myfile> is a convenient way of updating the customized source file. It is probably good practice to generate files into a temporary directory, customize them, move them to a target directory, and then having the ant scripts or makefiles handle the update process.

Grammar attributes that are declared "public" will have globally visible get/set accessors. Sometimes, attributes from one pass will need to be carried over to the next—symbol tables are good candidates—and public accessors make this possible.

Action Templates

A major goal of Yggdrasil has been to remove actions from tree construction to achieve target language independence. Having a syntactic interface to StringTemplate removes the need to generate target code through actions—a very important step. It is still not enough. For a large class of applications, the purpose of language processing is to build internal data structures, data structures which are then operated on by the application. For these, some form of actions seem necessary.

One way of achieving target independence is to implement a procedural language within ANTLR. That has never been a particularly attractive solution; one obvious side-effect would be to dramatically increase the size of the ANTLR runtime code. That would be a severe hindrance to porting the ANTLR runtime to a wide variety of target languages.

To avoid this, Yggdrasil is adopting a template approach to actions. Instead of including a behavioral syntax in Yggdrasil, actions are simply templates, templates which can reference both externally defined templates and grammar attributes. Target language code is then relegated to template files, template files which can be recoded for different targets.

The syntax for templates is

// ACTION or SEMPRED equivalent

action : "{" (QUOTED_STRING | ID) (":" (assignment)+)? ("}" | "}?") ;

assignment : ID "=" attribute_expression "," ;

QUOTED_STRINGs are anonymous templates, ID is a named template within the StringTemplateGroup assigned to UserLibrary (grammar option).

How do action templates differ from template attribute variables? This is a binding time issue. Action templates instantiate actions in the Yggdrasil-generated code; template attribute variables are used by the generated application to generate code.

Action templates effect Model/View/Controller separation: a properly templated grammar can be developed for one target environment and ported to another by simply editing template files. For anyone who suffered through debugging ANTLR 2 C++ code (Ric Klaren did an amazing job providing good C++ support, but C++ debugging was exceedingly unpleasant because the debuggers found C++ templates remarkably opaque and passed that opacity back to the often confused developer), this is a ray of sunshine. Applications can be developed for a friendly target language like Java and converted to a less friendly target by replacing template files.

Unfortunately, this forces the early incarnations of Yggdrasil to depend on the StringTemplate library. This increases the effort of targeting ANTLR and Yggdrasil to new languages, and adds to the amount of code packaged with the generated application. In later incarnations, I expect templates to be compiled into classes with the templates converted to arrays of objects (toString() then just iterates though the array, calling toString() for each of the objects in the array) and get/set methods generated for filling in the "holes". That should improve memory use, allow applications to be shipped without including sets of proprietary template files, and provide some (probably slight) speed improvements.

Lexical annotation

Yggdrasil supports the use of "!" in lexer rules without sacrificing performance in most cases. For strings in which "!" is used to remove leading and trailing characters, the begin and end pointers are adjusted accordingly. For strings in which internal characters are removed, a "copy on write" approach is taken: when an incipient gap is detected, characters are copied to an internal "copy" array, and the start and end positions of text in the token reference the copy array.

I intend to provide string and character attribute support in later versions; the ANTLR 2 lexer grammars were very powerful in part because of string editing support and the inclusion of attribute support should make it "easy" to write smart editors (the engine, not the display).

Keywords

In some languages, keywords (and symbols) are reserved and can appear in no other identifier. In others, keywords are special only in context: in a statement, for example, "if" might occur as a keyword -- `if (foo) bar;` -- or as a variable: `int if = 0;`. Many of the challenging translation problems deal with "languages" which do not have reserved keywords. ANTLR 3's DFA-based recognition of reserved strings represents a huge performance improvement over the ANTLR 2 literal table, but only addresses the problem of reserved keywords.

Languages with non-reserved keywords were a problem for ANTLR 2 because of the lack of the semantic predicate hoisting that was introduced in PCCTS (ANTLR 1). ANTLR 3 has brought back predicate hoisting, so that it is once again possible to do

```
{LT(1).getText().equals("foo") }? "foo".
```

This, however, does not go far enough. Yggdrasil supports both the use of string literals as reserved keywords or as context-dependent keywords through a grammar-level option. When the "ReservedKeywords" is set to "false", `"keyword"` is internally interpreted as

```
{LT(1).getText().equals("keyword") }? .
```

so that any keyword can be recognized in context, regardless of token type.

Another not yet implemented feature; I expect to support this in the second (0.6) release.

Multi-token lexing

Yggdrasil supports multi-token lexing. "protected" rules produce tokens, just as the default "public" rules do, but they are not included in the master case statement (or case statements, when lexical modes are supported). For example, consider

```
FOO
        :
        'A'  'B'  'C'  BAR 'g' 'h'
        ;
protected
BAR :  'D' 'E'  ;
```

BAR tokens are only recognized within the context of FOO; that is, the sequence of characters ABCDEgh is recognized as

FOO BAR FOO

while an isolated occurrence of DE in the input stream is an error (unless handled elsewhere in the grammar).

Multi-token lexing will probably be supported at about the same time as template compilation, largely because I see using it in constructing the template recognizer within ANTLR Yggdrasil. The lexical attribute support will include support for sub-rule tokens and the construction of tokens with rearranged text: that is a natural analogue of "labeled sub-rule" isolation of ^ context.

Development Plan

0.5: First release.

0.6: "Yggdrasil in Yggdrasil". Removes dependency on ANTLR 2.7 runtime; should include lexer attribute and editing support.

0.7: Initial tree grammar generation. May not support all "^ in loop" cases.

0.8: Adaptation to support generation of C code. Supporting languages that support classes and objects should be easy. For C, however, the translation of attribute types will be a slight challenge.

0.9: Documentation and cleanup. Will probably include "compiled" StringTemplates.

1.0: Production release.