An Attribute Unification Proposal for ANTLR

Loring Craymer

In the course of thinking about an attribute syntax for tree translocation support, I have devised an attribute approach that seems to unify various ideas that have been discussed on the mailing list and elsewhere. The features of this approach are

1.) Attributes have the following types:
    a. Atomic attributes
        i. String (default)
        ii. Integer
        iii. Real (double)
        iv. Boolean (optional: for semantic predicates)
        v. Carrier (Token or AST node)
        vi. User-defined (via action syntax)
    b. Compound attributes
        i. Payload
        ii. Queue
        iii. Table (associative lookup)
        iv. User-defined variants.
2.) Attributes exist at two levels
    a. Within a Payload attached to a Carrier
    b. At the Grammar level (compound attributes only)
3.) Fields representing compound types are references (pointers). This makes it possible to construct complex navigational structures during ANTLR tree walks.
4.) Rule-level and subrule-level scoping is achieved through grammar-level attributes. These attributes are associated with a rule via an "attributes { … }" expression. On entering the rule, the old attribute container is saved and a new one assigned. On exit, the old container is restored. [This implements Mitchell's idea of named attribute containers and implements dynamic scoping, although not quite as originally proposed.] Grammar-level attributes are non-null only if they are specified as attributes of at least one "active" rule.
5.) Attributes may be referenced (@a) or assigned to (@a = foo)
    a. Payloads may be assigned to any atomic attribute (and coerced in the process)
    b. An atomic attribute may only be assigned to another atomic attribute of the same type—neither typecasting nor coercion is supported.
6.) User-defined atomic types – declared as "Atomic { <target type> } X"—conforms to the access protocol above. That is @x is an X (assuming that x was defined as an X) and @x = @foo is allowed for Payloads and X's only.
    a. A reference is interpreted as get<Attribute>() in the target language: @foo becomes getFoo().

    b. Similarly, @foo=x assignment is interpreted as setFoo(x); setFoo is polymorphic and has a Payload and an X version.
7.) Payload and other type classes are defined in "types { … }" constructs. Any attribute type may be used to define a Payload field.
8.) Queues
    a. have two atomic attributes:
        i. first (@q.first = foo inserts foo as the first element of the Queue)
        ii. last (@q.last = foo appends foo to the Queue)
        iii. When referenced, elements in the queue are removed from the queue. (@foo = @q.first assigns and deletes the first element from the queue).
    b. @q (q is a Queue) returns all elements in a Queue. This would normally be used for tree translocation, but might also be useful for output grammars.
9.) Tables implement associative lookup. Table attributes are referenced through keys (Strings).
10.)Queues and Tables are treated as being homogeneous
    a. Declared with syntax analogous to Payloads: Queue X { Integer } declares that X objects are Queues of Integers.
    b. Data stored need not be homogeneous—subclasses are supported—but all manipulations conform to the type of the Queue or Table.
11.)User-defined types may have
    a. Computed attributes. Same translation as for user-defined atomic attributes: @x.b is interpreted as x.getB().
    b. "Set-only" attributes. (These would provide "context" for computed attributes).
12.)ANTLR 2 style arguments and return values are handled through attribute references and assignments.
13.)Tokens, AST nodes, and rules may be assigned to Carrier or Payload attribute fields. Payload assignments assign the Payload attached to the Carrier. Carrier assignments remove the Carrier from the rule construction stream—the Carriers will be inserted elsewhere in the tree.
14.)User-defined types are defined in "types" constructs with syntax "<Type> <type name> { element types/fields } <user type name>.
    a. This makes the element fields visible to ANTLR for warning messages.
    b. Multiple types may be defined in a single definition: in that case <user type> becomes <user type 1>, <user type 2>, etc.
    c. <user type name> is assumed to be implemented as a subclass of <type name>.
    d. Note that this is a means of adding computed attributes to the built-in types.
15.)Predefined fields
    a. Carrier: type

b. Payload: text *and* type
c. Queue: first *and* last

16.)Attributes are first class ANTLR elements. Syntactically,

a. @foo = @bar may occur in line without affecting token matching or tree structure.
b. @foo = bar mixes recognition of bar with assignment to foo.
c. For grouping, @{ foo = bar; bar = baz; } can also be used.

An example

```
class AttributeExample extends Parser;
types {
        Payload P1 {
                Integer x;
                Real y;
                String s;
        };

        Queue Q { Payload } Quser;

        Payload P2 {
                Carrier c;
                Payload p;
                Integer i;
        } P2user;
}

attributes {    // grammar level attributes
        P1 p;
        Quser q;
        P2user p2;
}

top :
        attributes { p; q; p2; }
        @p2.p = A       // store a copy of the payload
        B
        ^{ @p2.p }      // insert a copy of the A node
        subrule
        @p2.c           // translocation
        @i = p2.whatever        // computed attribute
        ;

subrule
        :
```

```
C
@p2.c = D
E
;
```

The corresponding tree grammar would be the rule

```
treetop
    :
    A B A C E D
    ;
```

This should give a bit of the flavor of using this version of attributes.

Attributes have data semantics

Attributes have only data semantics, not computational semantics. Computational semantics are restricted to actions; computation is not within the domain of ANTLR processing. The no typecasting and coercion only from Payloads feature stems directly from this restriction.

This restriction of attribute use drastically simplifies the implementation costs for an attribute class. Only three methods need be supported for a given atomic type, Type:

- Type getAttribute()
- void setAttribute(Carrier c)
- void setAttribute(Type t)

An elegant side effect of the extreme simplicity of this attribute interface is that computed attributes can be directly supported via getComputedAttribute() methods: @foo.c is translated to foo.getC(), so supporting computed attributes for user-defined types is essentially free. Similarly, "set-only" attributes are also possible

Scoping

The scoping model described derives from Monty's original dynamic scoping proposal, modified by John Mitchell to have named grammar-level attributes. The major change from the original proposal is that attribute container names are explicitly referenced as opposed to seeing an attribute name as belonging to one of the rule scopes currently active. I have taken this a bit further to have nestable scoping at either the rule or subrule level.

Compound types

Payloads were the original compound type-a neat way of packaging attributes for Tokens or AST nodes. One advantage of Payload classes is that field names are bound to fields

at compile time for maximal runtime efficiency. Another is that Payloads can reference other Payloads—complex navigational structures can be constructed from Payloads. Consider the example of a linked list Payload:

```
Payload DoublyLinkedList {
        DoublyLinkedList next;
        DoublyLinkedList prev;
}
```

This flexibility makes it possible to define and link a complex navigational data structure in tree walks without resorting to actions.

Queues deal with the problem of accumulating attributes from loops:
(A b @q.last = C)+
in either the order that they are encountered or in reverse order
(A b @q.first = C)+

Symbol tables are semi-ubiquitous in HLL translators. That provides sufficient reason to include the Table type, but this is offset by the implementation cost for generic Tables. I include Tables for two reasons: 1.) arbitrary symbol tables can be constructed from keys and compound attributes, and 2.) John Mitchell would undoubtedly complain loudly if I left them out.

User-defined features

I discovered the limitations of compiler-generated classes in dealing with JAXB. Generated classes are in general not sufficient; computational methods are application dependent. Fortunately, ANTLR need only deal with data semantics, so computational support can be implemented in the target language without requiring visibility to ANTLR. By allowing user-defined extensions to ANTLR-generated attribute classes to be used by ANTLR, grammars can have minimal or dependence on the target language.

Extension: Semantic predicates

While "attributes have data semantics only" removes most target language dependence, semantic predicates are still an annoying exception.

Semantic predicates query 1.) semantic state extracted from the input processed to date, and 2.) external state. The Boolean attribute provides an interface for evaluating state; much of the complex predication should be handled by computed Boolean attributes, The simple cases, though, require comparison operations: "==", "!=", ">", ">=", "<", and "<=". "&&" and "||" conjunctions are also desirable, as is "null" for use in comparisons. Are arithmetic operations required? I suspect not—computed attributes should provide workarounds in all cases—but I could be wrong. At any rate, I see expressing semantic

predicates in an attribute syntax rather than a target-language syntax as a desirable step forward.

Implementation costs

None of the proposed features are difficult to implement. Code generation can be very simple: get/set interfaces to fields, direct translation of type names, and the like. Only if a semantic predicate syntax is implemented is there noticeable processing, and even there it looks like code generation should be straightforward.