Yacc Meets C++

Stephen C. Johnson
Ardent Computer Corporation

ABSTRACT: The fundamental notion of attribute grammars [Knuth 1978] is that values are associated with the components of a grammar rule; these values may be computed by *synthesizing* the values of the left component from those of the right components, or *inheriting* the values of the right components from those of the left component.

The Yacc parser generator, in use for over 15 years, allows attributes to be synthesized; in fact, arbitrary segments of code can be executed as parsing takes place. For the last decade, Yacc has supported arbitrary data types as synthesized values and performed type-checking on these synthesized values. It is natural to think of this synthesis as associating a value of a particular type to a grammar symbol when a grammar rule deriving that symbol is recognized.

Languages such as C++ support abstract data types that permit functions as well as values to be associated with objects of a given type. In this framework, it appears natural to extend the idea of computing a value at a grammar rule to that of defining a function at a rule. The definition of the

Much of this work was done when the author was employed by AT&T Information Systems. A previous version of this paper was presented at the April 1988 EUUG meeting (London).

function for a given object of a given type depends on the rule used to construct that object.

In fact, this notion can be used to generalize both inherited and synthesized attributes, unifying them and allowing even more expressive power.

This paper explores these notions, and shows how this rule-based definition of functions allows for easier definitions and much more flexibility in some cases. Several examples are given that are hard to express using traditional techniques, but are naturally expressed using this framework.

1. Introduction

When we write a grammar rule such as

```
expr : expr '+' term ;
```

we frequently associate values with the components of the rule and use these values to compute the values of other components. For instance, in the above example we might wish to associate integer values with the symbols *expr* and *term*, and include a rule for generating the value of the left-hand *expr* from the value of the right-hand *expr* and *term*. Values associated with the left side of a rule that are computed from the values on the right are called *synthesized* attributes.

In other cases, we wish to use values associated with the left side of the rule to compute values of components on the right side of the rule. These values are known as *inherited* attributes. One important example of inherited attributes involves passing symbol table information into expressions and statements.

Allowing inherited and synthesized attributes to be specified without restriction makes it difficult to generate parsers automatically from the rule descriptions. For example, it is very difficult even to decide whether the definitions are potentially circular [Jazayeri et al. 1975]. Nevertheless, attribute grammars are very expressive, making them attractive both as a basis for editors and compilers [Reps 1984] and as a platform for more extensive computational and database systems [Horowitz & Teitelbaum 1985].

In a parallel development, languages such as Ada™ [United 1983], Modula-2 [Wirth 1983], and C++ [Stroustrup 1986] have explored ways of extending traditional program language type structures in much the same fashion. In Ada and C++, the type concept is extended to include not just the values and the data but also the actions that may be performed on these values. The key idea is that an abstract model of the data type, and the operations provided on it, is presented to the programmer; the details of the implementation are hidden.

Yacc is a parser generator that has been available under UNIX since the early 1970s. The original versions of Yacc permitted only one attribute, of integer type, for each grammar symbol. Later versions allowed other types, including structures, to be computed. However, because the parsing method is bottom up (LALR(1)), and the actions are executed as the rules are recognized, only synthesized attributes can be handled directly. More complex translations must be done by building a parse tree, and then walking this tree doing the desired actions.

In Yacc, a unique datatype may be associated with each nonterminal symbol. In this case, every rule deriving that nonterminal must return a value of the defined type. Each token may also have a defined type; in this case, the values are computed by the lexical analyzer. An action computes the value associated with the left side of a rule; this action depends on the particular rule and the values of the components on the right of the rule.

In trying to extend Yacc to handle C++, these two streams naturally came together in a prototype tool called y++. Since there was already an association of types with nonterminals, it became natural to ask whether functions could be defined on these types as well, and what meaning this might have. Some examples proved compelling.

1.1 Examples

Given a rule

```
expr : expr '+' expr ;
```

we might choose to define a function *print()*, on the nonterminal/type *expr*. In the context of this particular rule, *print()* might be defined as

```
print() { $1.print() ; putchar('+') ; $3.print() ; }
```

(As with Yacc, we will use `$1`, `$2`, etc. to refer to the components of the right side of the rule, and `$$` to refer to the left side).

We might also choose to define another function, *type*, on *expr*, and this might have a totally different definition:

```
type() { return( exprtype('+', $1.type(),
       $3.type() ) ) ; }
```

Since tokens are only created by the lexical analyzer (and never by rules), functions such as *print* and *expr* can be called implicitly as part of a particular rule, and need no special definition mechanism.

By allowing these rule-defined functions to have arguments and return values, we get many of the effects of inherited attributes:

```
type(TYPE t) { $1.type(t) ; $3.type(t) ; }
```

In C++, a function that is defined as a member of a class can obtain access to the values in an instance of that class; the keyword *this* allows such values to be explicitly manipulated. In y++, when a function *f* is defined on a nonterminal/type *X*, not only the value, but also the function definition itself depends on the rule used to define the instance of *X*.

A nice example is given by the rule:

```
expr : expr '+' expr ;
```

on which we define two functions, *polish* and *revpolish*:

```
polish() { putchar('+') ; $1.polish() ;
        $3.polish() ; }
```

```
revpolish() { $1.revpolish() ; $3.revpolish() ;
              putchar('+') ; }
```

These two functions can be similarly defined for other expressions. Two input line types can be defined as well:

```
line : "PRE" expr
line : "POST" expr
```

and a function, *print()*, that is defined as

```
print() { $2.polish() ; }
```

on the first rule, and

```
print() { $2.revpolish() ; }
```

on the second. Then, after a *line* has been recognized, *print* will print the expression in either prefix or postfix Polish form, depending on the initial keyword of the line.

The attribute grammar approach to this would require generating and storing both the prefix and postfix translations, and many intermediate translations as well. The above technique saves both space and time.

To summarize this section: we associate datatypes (C++ classes) with nonterminal symbols and tokens. In addition to values, these types have functions associated with them whose definition depends on the rule used to recognize a particular instance of the type. This mechanism generalizes both inherited and synthesized attributes; later sections discuss implementation and other applications.

2. Implementation

We have prototyped a tool, y++, to explore the semantic and syntactic implications of these ideas. Some features of y++ are:

1. Every grammar symbol, token and nonterminal alike, is associated with a C++ class.
2. Every class has 0 or more values, and 0 or more functions, defined on it.

3. Every grammar rule may have associated with it 0 or more functions that may be invoked to access and change the values accessible to that rule. These functions may access and change values of the result (left side) of the rule, and the components (right side) of the rule, and invoke other functions defined on the components of the rule.

In practice, y++ specifications are transformed to C++ programs and compiled. *yyparse* returns a value that is, in effect, a pointer to the parse tree. After calling *yyparse* (which causes some input to be read and parsed), the returned value is used to access the functions that are defined on the start symbol; presumably, these cause transformations and output to be done.

When *yyparse* is called, it creates a data structure that represents the parse tree. For tokens, it creates space large enough to hold the values, if any, included in the token. For nonterminals, the space created depends on the rule used to create the particular instance of the nonterminal. The rule

```
A : B C D ;
```

would cause space to be allocated as follows:

```
integer rule number
space for the A values
pointer to the B value
pointer to the C value
pointer to the D value
```

In the case of simple actions, not depending on the rule, we simply index past the value to obtain the components. In the case where the actions depend on the rule number, we generate a conditional based on the stored rule number. In the case where *B*, *C*, or *D* is a token, the value returned from the lexical analyzer is saved instead of a pointer.

There are a number of scope issues not yet resolved. If there are two calls to *yyparse*, for example, does the second parse tree overwrite the first, or do both remain active? The issue of default actions and values is also tricky. There is little point in wasting space on characters and literal keywords returned by the lexical analyzer when these are implicitly known from the rule number; the question is how to recognize this and exploit it.

A similar issue is the treatment of default functions. If a function is called for a rule that contains no definition for that function, should a default definition be assumed? We currently consider this a semantic error and produce a message, *semantic error*, by analogy with the *syntax error* message of Yacc.

Another issue is error handling. There is a premium in being able to return a sensible structure for any input, even those in error, to allow the user to craft special functions that give particularly good error messages. The exact mechanism by which these *error rules* might be constructed is still open.

Finally, given a data structure representing a parse tree, it is very nice to be able to rewrite it; y++ should probably provide such operations through a user interface.

2.1 A Simple Example

This section sketches how y++ can be used to make a preprocessor that translates extensions into a base language. We begin with a function *ident*, defined on every nonterminal symbol of the base language grammar; this function, when called, produces a literal text representation of the rule. For example, for the rule:

```
stat : expr ';'
```

we might define *ident* as

```
ident() { $2.ident() ; putchar(';') ; }
```

This grammar can quickly be extended to a preprocessor by simply adding rules for the new constructions, and defining *ident* on the new rules to translate into the base definition. For example, suppose we wish to augment C with the *forever* statement. After recognizing the keyword in the lexical analyzer, we add the rule:

```
stat : "forever" stat ;
```

and the associated definition of *ident*:

```
ident() { printf("while(1)") ; $2.ident() ; }
```

Clearly, translators that required symbol tables, etc., would be harder, but one could envision a standard C grammar and lexical analyzer being far more reusable in y++ than in Yacc.

2.2 Impressive Example

Giegerich and Wilhelm have discussed the difficulty of generating "short-circuit" evaluation of Boolean expressions using the usual forms of syntax-directed translations (see also Aho et al. [1985]). This becomes relatively straightforward in y++. A function, *bool_gen(t, f, n)*, is defined on the rules involving the short-circuited operators. *t* is the label to go to if the expression is true, *f* the label for false, and *n* has the "preferred" label, either *t* or *f*. The rule

```
expr : expr OR expr
```

for example would define *bool_gen* as

```
bool_gen( t, f, n )
{
   int x = get_new_label();
   $1.bool_gen( t, x, x );
   define_label( x );
   $3.bool_gen( t, f, n );
}
```

and similarly for AND and NOT. The definition for those expressions not involving short-circuit operations would look like:

```
bool_gen( t, f, n )
{
   . . . /* get the value of the expression */
   if( t == n )
      { . . . /* branch if false to label 'f' */ }
   else
      { . . . /* branch if true to label 't' */ }
}
```

3. Conclusion

This paper describes a simple extension of parser generators to handle abstract data types; in this way, some translations can be specified easily that would be more difficult to describe with conventional attribute grammars.

Moreover, the notions seem to generalize attribute grammars, while at the same time allowing the ideas of Yacc to be brought to bear on the concepts in C++, or perhaps *vice versa*.

References

A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, New York, 1985.

Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, MA, 1986.

R. Giegerich and R. Wilhelm, Counter-one-pass features in one-pass compilation: a formalization using attribute grammars, *Information Processing Letters* 7(6) pages 279-284.

S. Horowitz and T. Teitelbaum, Relations and Attributes: A symbiotic basis for editing environments, *Proceedings of SIGPLAN 85 Symposium on Language Issues in Programming Environments*, Seattle, WA, pages 93-106, June 1985.

M. Jazayeri, W. F. Ogden and W. C. Rounds, The intrinsic exponential complexity of the circularity problem for attribute grammars, *Communications of the ACM* **18**(12) pages 697-706, 1975.

D. E. Knuth, Semantics of context-free languages, *Math. Syst. Theory*, **5**(1) pages 95-96, March 1971.

T. Reps, *Generating Language-Based Environments*, MIT Press, Cambridge, MA, 1984.

United States Department of Defense, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983, Feb. 1983, Springer-Verlag, New York.

N. Wirth, *Programming in MODULA-2*, Springer-Verlag, New York, 1983.

[*submitted April 16, 1988; accepted May 23, 1988*]