Parsing with C++ Deferred Expressions

Damian Conway

Victorian Centre for Image Processing and Graphics
Department of Computer Science
Monash University
damian@bruce.cs.monash.edu.au

Abstract

A technique for constructing embedded grammar-based parsers in C++ was described in [1]. This paper presents an extension to that technique which allows parsing actions to be specified directly as part of the grammar, rather than indirectly (as function calls).

1. Introduction

The `Parser` class library[1] defines a set of related classes which may be used to build parsing grammars within a C++ program. Extensive use of operator overloading enables these grammars to be specified with a syntax very similar to the familiar *yacc* format. The library includes features such as lazy evaluation of terminals, user-defined resolution of shift-reduce and reduce-reduce conflicts, polymorphic value assignments to terminals and non-terminals, context-sensitive parsing, and the embedding of actions within a parser.

The library as presented in [1] has one major limitation – the specification of actions to be performed during parsing is restricted to embedded pointers to functions. When a parsing rule is successfully matched, these function pointers (suitably encapsulated in `Action` class objects) are dereferenced and their corresponding functions called.

Because function definitions in C++ cannot be nested, the use of function pointers to specify parser actions necessarily implies that the code corresponding to the actions must be physically separate from the grammar in which those actions are used. Although it produces clean and modular code, this style of building parsers is somewhat tedious. More importantly, the resultant embedded grammars are considerably more difficult to understand than the equivalent *yacc* specification.

This paper introduces an additional set of classes which implement a C-like interpreted language suitable for specifying parser actions. Statements in the language are converted to an expression tree encapsulated in a class object. Such "expression" objects can then be included as part of an embedded grammar.

2. The Deferred Expression Idiom

Van Wyk [2] describes a technique for building a set of classes which encode and solve simultaneous equations specified in a declarative manner within a C++ program. The technique overloads common operations such as addition, subtraction and equality test, causing such operators not to perform the corresponding operation, but to generate an object which represents the operation. Such representative objects can then be passed to a suitable modified constraint solver [3] in which the operations they represent may be evaluated at need.

In a similar manner, the classes described in this paper used overloaded operators (in this case, virtually all of the operators in the C++ language) to generate an object representing the parse tree of the expression. This expression object belongs to a class derived from the root `Parser` class, and hence the original expressions used to generate it may be incorporated seamlessly into any embedded grammar built with the Parser library. Figure 1 illustrates the technique, components of which are described in Sections 3 through 8.

During parsing, expression objects (like the `Action` objects they supersede) are matched trivially against the input stream. That is, as part of a rule to be matched, expression objects are ignored and therefore consume none of the input stream.

Once a rule has been successfully matched, each of its components are evaluated by calling their virtual `Evaluate()` method, which must be added to the `Parser` base class. In order to initiate this match-evaluate sequence, a new method, `Parse()`, is added to the `Rule` class. This method attempts to match the grammar

---

[1]As described in [1] and available by anonymous ftp from `bruce.cs.monash.edu.au` in the directory `/pub/damian/Parser.shar.Z.uu`

rooted at its parent `Rule` against the input stream and, if successful, evaluates the resultant parser tree and returns its final value.

Terminals and rules within the grammar evaluate trivially (that is, their `Evaluate()` methods do nothing) whilst objects of class `Action` call the function associated with them. Calling the `Evaluate()` method of one of the new expression objects causes the entire expression tree contained in it to be evaluated, effectively executing the original expression embedded in the grammar.

```
#include "Parser.h"

main()
{

RegexToken      INT("[0-9]+");
RegexToken      IDENT("[A-Za-z_][A-Za-z0-9_]*");

Token           LSQ("[");
Token           RSQ("]");
Token           PLUS("+");

Rule            ArrayDecl;
Rule            ArraySize;

ParserOstream   CERR(cerr);

ArrayDecl =  IDENT + IDENT + LSQ + ArraySize + RSQ +
                    (
                     IF (S4 > 0)
                        ( CERR << "Valid length" << ENDL),
                     ELSEIF (S4 == 0)
                        ( CERR << "Dubious length" << ENDL ),
                     ELSE
                       ( CERR << "Invalid length" << ENDL ),
                     SS =  ( IF (S4 > 0) ( S4 ), ELSE (1) )
                    )
          ;

ArraySize =  INT +
                   ( SS = S1 )

          |  INT + PLUS + INT
                   ( SS = S1 + S2 )

          ;

int arrayLength = ArrayDecl.Parse(pstream(cin));
}
```

Figure 1: Use of deferred expressions in an embedded parser..

3. The `ParserExpr` Base Class

In order to simplify the construction and evaluation of expression trees, all classes representing deferred expressions are derived from a common base class, `ParserExpr`, which is in turn directly derived from the root class `Parser`. Hence `ParserExpr` objects can be placed in embedded parser specifications, just like `Token`, `Rule` and `Action` objects.

The `ParserExpr` class extends class `Parser` by adding two protected members, `lhs` and `rhs`, of type `ParserExpr*`. These members are used to implement the (predominantly binary) expression tree, although some derived classes add additional members to handle N-ary expressions such as cascaded *if*s (see Section 8). The `ParserExpr` class provides a constructor which initializes its two members (to zero, by default). The constructor is protected, ensuring that the base class itself is never instantiated.

Classes corresponding to useful operations can then be derived from the `ParserExpr` class, by redefining their virtual `Evaluate()` methods. This method computes and returns the value of the expression rooted at the current object. See Sections 5 through 8 for examples. Note that the virtual `Match()` method of each derived class can be inherited from the `ParserExpr` base class without redefinition, because `ParserExpr::Match()` is defined to match any input stream trivially (that is, simply to return the value of the object's `this` pointer.)

Instances of derived expression classes are created within a grammar by defining the appropriate overloaded operators (again, see Sections 5 through 8 for examples).

```
class ParserExpr : public Parser
{
protected:
       ParserExpr * lhs;
       ParserExpr * rhs;

       ParserExpr(ParserExpr * left = 0, ParserExpr * right = 0)
             : lhs(left), rhs(right)  {}

public:
       virtual Parser * Match(Context c)
             { return this; }
};
```

Figure 2: A base class for representing deferred expressions.

4. Embedding References To Grammar Components

The basis of most useful parser actions is the manipulation of values matched by individual terminals or rules during the parse. To this end, parser specification languages typically provide a mechanism for referring to the current value of individual components of a rule. In the *yacc* specification language, for example, these references are known as `$1`, `$2`, etc., and correspond to the values of first, second, etc. components of a right hand side of a rule. The left hand side of a rule can also be referred to, as `$$`.

To create such references within the deferred expressions described in this paper, a special class, `ParserVar`, is derived from class `ParserExpr`. A `ParserVar` object contains an integer member, `index`, indicating the ordinal position within the current parent rule of the component being referred to. When the `Evaluate()` method of such an object is called, it interrogates the `Context` object passed as its argument and returns the value of the appropriate component of the parent rule. This is equivalent to evaluating `$index` in a *yacc* specification. Figure 3 illustrates the necessary class definition.

```
class ParserVar : public ParserExpr
{
private:
   int index;
public:
   ParserVar(int i) : index(i)    {}
   ParserVar(Value v) : index(-1) { this->Data = v; }

   virtual Value & Data(Context c)
       {
         if (index>0)  return c.Parent->Component(index)->Data;
         if (index==0) return c.Parent->Data;
         if (index<0)  return this->Data;
       }
};
```

Figure 3: A class for representing grammar component values.

Two values of the member index are reserved for special purposes. If index is zero, the ParserVar refers to the value of the parent rule of the current context, and hence is equivalent to $$ in a *yacc* specification. If index is negative, the ParserVar object represents a literal value, which is stored in the Data member (inherited from class Parser) of the object. The capacity to store values literally (as well as by reference) is critical, as it permits expressions containing literal values to be embedded in a grammar. Class ParserVar provides a special constructor, which takes a literal value as its single argument, stores it in its Data member, and automatically sets index to -1.

It is convenient to predefine certain frequently used instances of class ParserVar in order to simplify the construction of appropriate grammars. As the use of the dollar sign as the first character of an identifier is not permitted in the proposed ISO/ANSI C++ standard, in the Parser class library the character 'S' has been used to replace it, yielding SS for $$, S1 for $1, S2 for $2, etc. Figure 1 illustrates the use of these globals in deferred expressions.

5. Embedding Deferred Assignments

Experience indicates that the single most common expression embedded in a Parser grammar is the simple assignment, such as: SS = S1 (which assigns the value of the first component of a rule to the rule itself; effectively passing a value back up the parse tree.)

This deferred assignment must be carried out when the rule has been successfully matched and is being evaluated. Hence the assignment operator of class ParserVar is overloaded as shown in Figure 2. Note the overloading for an argument of type Value. This enables literal values to be used as the assigned value in a deferred assignment, once they have been suitably encapsulated in a ParserVar object.

```
ParserAssignment & ParserVar::operator=(ParserExpr & pe)
{
       return * new ParserAssignment(this,&pe);
}

ParserAssignment & ParserVar::operator=(Value v)
{
       return * new ParserAssignment(this,new ParserVar(v));
}
```

Figure 4: A deferred assignment operator.

The ParserAssignment class is derived from ParserExpr and stores pointers to a ParserVar (the target of the assignment) and a ParserExpr (representing the value to be assigned) in its lhs and rhs members. When a ParserAssignment object is evaluated, it executes the deferred assignment, as indicated in Figure 4. As is standard in C and C++, the evaluation of the assignment returns the assigned value, so that deferred assignments may be chained or otherwise embedded in larger expressions.

```
class ParserAssignment : public ParserExpr
{
public:
       ParserAssignment(ParserVar * lhs, ParserExpr * rhs)
             : ParserExpr(lhs,rhs)  {}

       virtual Value Evaluate(Context c)
             { return lhs->Data(c) = rhs->Evaluate(c); }
};
```

Figure 5: A class for representing deferred assignments.

Similar classes representing other deferred assignment operations can easily be created by adding the corresponding overloaded operator to the ParserVar class, creating the corresponding deferred assignment class, and redefining the Evaluate() method of the new class to perform the appropriate operation.

6. Embedding Other Binary Operations

The same approach can be used to create other types of deferred binary operators by creating suitable expression classes (for example, deferred addition as presented in Figure 6.) Note the structural and functional similarity between the `ParserAddition` and `ParserAssignment` classes. In fact, the only difference between the two is that `ParserAddition` objects are created using a global overloaded operator (since such operations are not required to be class methods, as is the overloaded assignment operation.) Again notice the two versions of the operator with literal arguments.

```
class ParserAddition : public ParserExpr
{
public:
       ParserAddition(ParserExpr * lhs, ParserExpr &* rhs)
             : ParserExpr(lhs,rhs)  {}

       virtual Value Evaluate(Context c)
             { return lhs->Evaluate(c) + rhs->Evaluate(c); }
};

ParserExpr & operator+(ParserExpr & pe1, ParserExpr & pe2)
{
       return * new ParserAddition(&pe1,&pe2);
}

ParserExpr & operator+(ParserExpr & pe1, Value v)
{
       return * new ParserAddition(&pe1,new ParserVar(v));
}

ParserExpr & operator+(Value v, ParserExpr & pe2)
{
       return * new ParserAddition(new ParserVar(v),&pe2);
}
```

Figure 6: A class for representing deferred additions.

7. Embedding Deferred Output

Yet another common action during parsing is output. In C++, I/O is handled via overloaded binary operators. The paradigm is easily reproduced for deferred expressions by creating two new classes (see Figure 7). This section presents an implementation of deferred output (the more common operation required during a parse) but the technique is readily adapted to implement deferred input as well.

The first additional class required is `ParserOstream`, which is used to encapsulate ostream objects[1] using a private reference. Note that `ParserOstream` need not be derived from `ParserExpr`. However doing so allows a `ParserOstream` object to be used as a test condition in a deferred selection construct (see Section 8 below).

The second necessary class, `ParserOutput`, is used to represent complete deferred output operations and must be derived from class `ParserExpr`. A `ParserOutput` object stores a reference to an `ostream` and a list of pointers to `ParserExpr` objects, whose values are to later to be evaluated and written to the `ostream`.

A `ParserOutput` object is created when the overloaded left shift operator (`operator<<`) of a `ParserOstream` object is called with an argument of type `ParserExpr`. An overloaded left shift operator is also added as a method of class `ParserOutput`, to permit chaining of output operations (as demonstrated in

[1]In fact, this class is not strictly necessary, as the global left shift operator could be overloaded for arguments of type `ostream&` and `ParserExpr&`. However, such an approach would preclude the use of literal values as the second argument in a deferred output operation, since `operator<<(ostream&,Value&)` is already overloaded to implement undeferred output of `Value` objects.

Figure 1). Each left shift operation causes a pointer to the second argument (a `ParserExpr`) to be added to the list in the `ParserOutput` object. The completed `ParserOutput` object is then included in a particular `Rule`, which is subsequently parsed. As with all other deferred expressions the `ParserOutput` object trivially matches during the parse phase and then performs the required deferred output when its `Evaluate()` method is called.

```
class ParserOutput : public ParserExpr
{
private:
       ostream & os;

       DynamicArray<ParserExpr*> expression;
       int last;

       ParserOutput(ParserOstream & pos, ParserExpr & expr)
             : os(pos.os), last(0)
             {  expression[last] = & expr;  }
public:
       virtual ParserOutput & operator<<(ParserExpr & expr)
             {
                   expression[++last] = & expr;
                   return *this;
             }

       virtual Value Evaluate(Context c)
             {
                   for (int i=0; i<=last; i++)
                         os << expression[i]->Evaluate(c);
                   return os.good();
             }
};

class ParserOstream : public ParserExpr
{
friend class ParserOutput;
protected:
       ostream & os;

public:
       ParserOstream(ostream & theos)
             : os(theos) {}

       virtual ParserOutput & operator<<(ParserExpr & expr)
             {  return new ParserOutput(os,expr);  }

       virtual Value Evaluate(Context)
             {  return os.good();  }
};
```

Figure 7: Classes to support deferred output.

8. Embedding Control Structures

One important binary operation that must treated slightly differently is the sequencing operation: `operator,()`. By itself, there is no reason that sequencing of expressions could not be accomplished simply by defining a global `operator,()` function that returns a `ParserSequence` object. However, it is also desirable to provide conditional sequencing – a deferred *if* statement – and this requires somewhat more sophistication from the comma operator.

Instead of being declared as a global function, the comma operator is defined as a virtual method of class `ParserExpr` itself (as in Figure 8). In addition, another overloaded sequence operator is defined, taking a second argument of class `ParserAlternative`. Objects of this class represent the alternative actions in a (cascaded) *if* construct. This additional comma operator simply produces an error message in the general case, but will be redefined to do something more useful in the `ParserIf` class presented below.

```
class ParserExpr: public Parser
{
              :
              :        // As before
public:
       virtual Parser & operator,(ParserExpr & pe)
         { return new ParserSequence(this,&pe); }

       virtual Parser & operator,(ParserAlternative & pe)
         {
             FatalError("ELSEIF or ELSE without preceding IF.");
             return *this  // Never executed
         }
};
```

Figure 8: The virtual sequence operator for ParserExpr.

In order to permit the use of selection statements within deferred expressions, a special class, `ParserIf`, is created. Unlike most of its sibling classes, `ParserIf` contains two dynamic arrays of `ParserExpr` pointers, enabling it to store a series of alternate conditions and actions. A helper class, `ParserAlternative`, is also required, but need not belong to the `ParserExpr` hierarchy. Figure 9 illustrates the necessary code.

Selection constructs are created using the global `IF()` function. This creates a `ParserIf` object and initializes its first condition to point to the function argument. The expression to be returned (if the deferred condition evaluates to non-zero) is then passed to the `ParserIf` object using the function call operator. This two step approach produces a syntax very similar to the standard C / C++ syntax (see Figure 1), except that brackets are used to contain the body of the deferred *if*, rather than braces.

Another significant difference is that a `ParserIf` object represents an expression rather than a statement, hence its result may be used in subsequent expressions (see Figure 1 for an example). Thus the deferred *if* also provides the functionality of the ternary operator (`?:`). Indeed, this particular situation is one in which seems to have been overlooked[1] by Stroustrup and Ellis in their prohibition [4] of an overloaded `operator?:()`.

Class `ParserIf` redefines the virtual sequence operator it inherits from class `ParserExpr`, so that any `ParserAlternative` object which follows a deferred *if* expression are appended to the list of possible condition / expression pairs.

`ParserAlternative` objects are created through the global function `ELSEIF()`, and contain pointers to a deferred condition and a corresponding deferred expression[2]. These pointers are appended to the respective lists in the preceding `ParserIf`. `ParserAlternative` objects may also be created using the global function `ELSE()`, in which case their condition pointer is set to zero, which the method `ParserIf::Evaluate()` subsequently interprets as "always succeed".

9. Conclusion

The techniques presented in this paper greatly improve the usability of the `Parser` classes presented in [1]. The approach illustrated can be extended to any binary or unary C++ operator which can be overloaded. Other extensions which might be considered include support for deferred function calls, other control structures (in particular, loops), and support for a dynamic record type (as an extension of the `Value` class).

---

[1]This is not intended as a criticism: this application of operator overloading is quite esoteric and was hardly likely to suggest itself to the hardworking language designers.

[2]Class `ParserAlternative` is structurally and functionally very similar to class `ParserIf`. Objects of class `ParserAlternative` are creating with a one argument constructor which stores a pointer to the deferred condition, and completed using their own `operator()` to store a pointer to the corresponding deferred expression.

The concept of deferred expressions also has wide application beyond this particular problem domain. Applications in the areas of lazy evaluation, extensible interpreters, and declarative or self-modifying styles of programming will readily suggest themselves to the reader.

```
class ParserIf: public ParserExpr
{
private:
      DynamicArray<ParserExpr*> condition;
      DynamicArray<ParserExpr*> expression;
      int last;

public:
      ParserIf(ParserExpr & cond)
          : last(0)
          {  condition[last] = & cond;  }

      Parser & operator()(ParserExpr & expr)
          {  expression[last] = & expr;  }

      virtual Parser & operator,(ParserAlternative & pa)
          {
             condition[++last] = pa.condition;
             expression[last] = pa.expression;
             return *this;
          }

      virtual Value Evaluate(Context c)
          {
             for (int i=0; i<=last; i++)
                if (!condition[i] || condition[i]->Evaluate(c))
                   return expression[i]->Evaluate(c);
             return 0;
          }
};

ParserIf & IF(ParserExpr & condition)
{
      return * new ParserIf(condition);
}

ParserAlternative & ELSEIF(ParserExpr & condition)
{
      return * new ParserAlternative(condition);
}

ParserAlternative & ELSE(ParserExpr & expression)
{
      return (new ParserAlternative(0))->operator()(expression);
}
```

Figure 9: Code to support deferred selection constructs.

References

[1] Conway, D.M., *Parsing with C++ Classes*, ACM SIGPLAN Notices, vol. 29, no. 1, pp. 46-52, January 1994.

[2] Van Wyk, C.J., *Arithmetic Equality Constraints as C++ Statements*, Software - Practice and Experience, vol. 22, no. 6, pp. 467-494, 1992.

[3] Derman, E. & Van Wyk, C.J., *A Simple Equation Solver and its Application to Financial Modelling*, Software - Practice and Experience, vol. 14, no. 12, pp. 1169-1184, 1984.

[4] Ellis, M, & Stroustrup, B., *The Annotated C++ Reference Manual*, §13.4 , Addison-Wesley, 1992.