Parsing with C++ Classes

Damian Conway

Victorian Centre for Image Processing and Graphics
Department of Computer Science
Monash University
Clayton, Victoria 3168
Australia
email: damian@bruce.cs.monash.edu.au

Abstract

Philip W. Hall IV [1] has outlined a preprocessor technique for building recursive descent parsers using C++ constructor functions. This paper presents an alternative C++-based parser generation scheme which uses a hierarchy of classes with appropriately overloaded operators to embed the actual grammar of one or more parsers in a standard C++ source file. This approach eliminates the need for preprocessing of source files and permits the use of parsing schemes somewhat more sophisticated than simple statically-defined recursive descent.

1. Overview of the technique

The technique uses a set of general-purpose parsing classes† , instantiations of which represent individual grammar rules, input tokens and parsing actions. For each class, a set of operators is defined which enables the programmer to write grammar declarations in a format not unlike that for the *yacc* preprocessor [2]. All of the parsing classes are based on a single class (**Parser**), from which they inherit a common interface.

To specify a grammar (see Figure 4 for a full example), the programmer first instantiates one or more objects of class **Token** (or one of its derived classes - see section 4 below.) Each **Token** object is used to recognize a single token from the input stream. **Token** objects tokenize the input stream lazily, so there is no separate tokenization phase in the parsing process. This creates more flexibility in handling ambiguous or context-sensitive grammars

Tokens are recognized according to the current context of the parse (that is: the current **Token** object being matched). Hence, at different times during the parse, a given set of characters in the input stream may be identified as representing different tokens. This capacity is useful in situations where a language to be parsed has non-reserved keywords, multiple name-spaces or is otherwise context-sensitive. The drawback of this approach is the cost of retokenization on backtracking. This cost may be substantially reduced by providing **Token** objects with some short-term memory, as described in section 4 below.

Once a set of **Token** objects has been defined, the programmer instantiates one or more objects of class **Rule** (or one of its derived classes - see section 5 below.) The grammar for each **Rule** is specified by assigning to it a production, which is specified as a C++ expression. Productions are composed of the disjunction of one or more sequences of **Rule** and/or **Token** objects. Naturally, a production may recursively reference the **Rule** object to which it is ultimately assigned.

Parsing of the grammar is initiated by passing a special input stream (see section 2 below) to a **Rule** object, which is usually the root of the grammar. This root object then attempts to recursively match its component production(s) against the input stream. A match is achieved when the input stream is successfully parsed by the entire sequence of **Rule**/**Token** objects in at least one of the root object's alternative productions.

Each **Rule** object provides a method which can be used to resolve ambiguous grammar specifications. Such ambiguities occur when two or more of the object's productions successfully match the input stream. The default behaviour for all **Rule** objects is to accept the production which matches the most tokens from the input stream. This is equivalent to choosing to shift in order to resolve a shift/reduce conflict. In the event that two productions match the same maximal number of tokens, the first production matched is se-

† Regrettably, there is as yet no uniform terminology for the object-oriented paradigm. In this paper, the term "class" is used to refer to a user-defined data structure and the term "object" for any instantiation of such a data structure. Functions encapsulated within a class are referred to as "methods" and encapsulated data fields or variables are called "members". Unless otherwise indicated, methods will be assumed to be public (that is, accessible anywhere in the scope of their containing object) and members will be assumed to be private (accessible only to the methods of their containing object.)

lected, which is equivalent to choosing the earliest reduction to resolve a reduce/reduce conflict. Other matching behaviours are easily specified (see section 7.)

2. A stream class to support parsing

The parsing model described in section 1 permits arbitrary lookahead and backtracking during a parse. To accommodate this, a special type of input stream (class **pstream**) has been constructed. Class **pstream** supplies the same standard input (**>>**) operators that other C++ stream classes offer, but includes two special methods:

Mark pstream::GetMark(void)
...which returns a indicator of the current read position.

void pstream::SetMark(Mark)
...which resets the current read position to the specified position.

Figure 1 illustrates the use of class **pstream**. In this example a **pstream** object is first bound to the standard input stream (**cin**) and then attempts to read in a sequence of three integers, without corrupting the stream on failure:

```
pstream ps(cin);
Mark origin = ps.GetMark();
if (!(ps >> x >> y >> z)) {
  ps.SetMark(origin);
}
```

Figure 1: The use of pstream

Note that this functionality is similar to that provided by the **tellg()** and **seekg()** members of standard C++ streams except that a **pstream** object may be bound to any other kind of C++ input stream (including an **istrstream**) and is guaranteed to correctly seek to any position in the resulting input sequence, whether or not the original input stream is buffered or otherwise capable of seeking backwards.

In addition, class **pstream** provides several methods which are particularly useful when building error-reporting into parsers, including methods to:

- return the current line number in the input stream,
- return a copy of the input yet to be read,
- return a copy of a particular **'\n'**-bounded line of the input stream,
- return a copy of the characters most recently consumed by a successful input (**>>**) operation (even if the value returned was a numerical type.)

3. The Parser base class

The abstract class **Parser** describes all the generic capacities required of any lexical token, grammar rule or embedded action. Specifically, classes derived from **Parser** must supply appropriate redefinitions of the following four methods:

virtual int Parser::NumberOfTokens()
...which returns the number of tokens matched by the object.

virtual int Parser::NumberOfChars()
...which returns the number of characters matched by the object.

virtual Parser * Parser::Copy()
...which returns a dynamically-allocated duplicate of the object. The copy is marked as a copy to facilitate garbage collection within **Parser** objects.

virtual Parser * Parser::Match(Context)
...which initiates an attempt to match the token, rule or action represented by the object. The **Context** argument is a data structure containing pointers to the **pstream** being parsed, to the current object and to its immediate parent object (if any).

The **Parser** class also provides the following standard methods to all inheriting classes:

Rule Parser::operator | (Parser&)
...which overloads the OR operation between any two **Parser** objects to return a **Rule** representing the disjunction of the two arguments.

Rule Parser::operator + (Parser&)
...which overloads the CONCATENATE operation between any two **Parser** objects to return a **Rule** representing a sequence of the two arguments.

In addition, it is usually desirable to provide members and associated methods which give **Parser** objects the following capacities:

- The ability to switch on and off run-time debugging statements which trace the matching behaviour of **Parser** objects.
- The ability to set global variables which signal conditions such as *abort-parse*, *terminate-matching-of-this-parser*, and *commit-to-current-disjunctive-branch*. These are particularly useful for creating appropriate embedded actions for handling fatal error and non-fatal errors and for permitting "cuts" in the parse tree.

4. Classes which represent tokens

Given the above interface for the base class **Parser**, it is relatively easy to define a family of classes whose function is to match terminal productions within the grammar. For example, consider a simple token class (**Token**), which matches fixed character strings within the input stream. The **Token** class has a member (**matchstring**) of type **char***, or better still of class **String**, which stores the fixed character string to be matched. This private member is initialized with a suitable constructor function.

Token objects inherit their interface from the **Parser** class and redefine the four common operations as indicated below. Note that composition of productions is independent of the nature of the components, so the "**+**" and "|" operators need not be overloaded for this (or indeed, any other) class.

int Token::NumberOfTokens()
...which returns 1

int Token::NumberOfChars()
...which returns the number of characters in **matchstring**

Parser * Token::Copy()
...which returns a dynamically-allocated duplicate of the **Token** object.

Parser * Token::Match(Context)
...which first records the current read position of the input **pstream** (supplied as part of the **Context** object) and then reads successive characters from the **pstream**, comparing them against the corresponding characters in **matchstring**. If there is a mismatch at any position, the **pstream** is immediately rewound to the recorded read position and the value zero is returned (indicating failure). If there is no mismatch before the end of **matchstring**, the match succeeds and the method returns a pointer to the current object.

The construction of more specialized token classes is similarly straightforward. A token class for matching regular expressions (class **RegexToken**) will require a private member of type **Regex**, and a suitable constructor. The **RegexToken::Match()** method is redefined so that it reads increasingly longer sequences of characters from the **pstream** and attempts to match them against the encapsulated **Regex**, returning zero on failure or a self-referential pointer on success.

On the other hand, a token class for matching integers does not require any additional members. The **IntToken::Match()** method is simply redefined to attempt to read a locally declared integer. Once again, the method returns zero on failure and a pointer to the object on success.

If the grammar is context-sensitive, it may be desirable to provide a class of tokens with a memory. Such a class (**MemToken**) uses a **char*** or **String** member, which is initially set to an empty string. On its first successful attempt to match (probably in a manner similar to **RegexToken::Match()**) a **MemToken** object records the character sequence it matched. Subsequent calls to that object's **Match()** method succeed only if the **pstream** furnishes an exact match for the remembered character sequence. The **MemToken** class also provides a **Forget()** method to reset the matching behaviour.

This idea of token memory is also useful in improving the efficiency of lazy evaluation under backtracking of the input stream. **Token** objects can be provided with two members (**lastmatchposition** and **lastmatchlength**), which record the position (a **Mark**) and extent (an integer) of the last successful match of the token on the current input stream. When a **Token** object attempts to match again, it first compares **lastmatchposition** with the current read position. If the positions are identical, the **Token** object short-circuits the matching process and returns a self-referential pointer, indicating success. This technique can be extended to **Rule** classes to further improve parser performance.

The construction of other, more elaborate token classes follows the same general principles outlined above and is left to the reader's imagination.

5. Classes which represent grammar rules

In a hierarchical grammar, terminals can be aggregated sequentially or disjunctively to produce non-terminal rules. The **Parser** class hierarchy provides a container class (**Rule**) for such aggregations and defines associated operators to facilitate their construction. As mentioned in section 3, the "**+**" operator is used to construct sequences and the "|" operator to construct disjunctions.

A given rule object may be used to store either a single sequence of **Parser** objects or a single disjunction of such objects. Hence two distinct types of rule class are defined: **SequenceRule** and **DisjunctionRule**. Composite rules (for example, a disjunction of sequences) are formed by creating a series of **SequenceRule** objects and storing them in a single **DisjunctionRule** object. Note that the default precedence of the "**+**" and "|" operators in C++ ensures that such composite objects are correctly structured.

The rule-type classes are derived from the **Parser** class by redefining the following member functions:

virtual int SequenceRule::NumberOfTokens()

...which calls the **NumberOfTokens()** method of each matching **Parser** object in the **SequenceRule** and returns the sum of these counts.

virtual int SequenceRule::NumberOfChars()

...which calls the **NumberOfChars()** method of each matching **Parser** object in the **SequenceRule** and returns the sum of these lengths.

virtual int DisjunctionRule::NumberOfTokens()
virtual int DisjunctionRule::NumberOfChars()

...which call the corresponding method of each **Parser** object in the **DisjunctionRule** and returns the maximal value found for any successful match.

virtual Parser * DisjunctionRule::Copy()
virtual Parser * SequenceRule::Copy()

...which returns a dynamically-allocated duplicate of the object.

virtual Parser * DisjunctionRule::Match(Context)

...which iterates through each branch of the disjunction, attempting to match it (by calling **Match()** methods.) When all branches have been tried, a pointer corresponding to the best component match is returned (see section 7.)

virtual Parser * SequenceRule::Match(Context)

...which sequentially calls the **Match()** method of each **Parser** object in the sequence. If any **Match()** fails, the whole sequence fails and the method returns zero. If all attempts succeed, the method returns a pointer to the **SequenceRule** object on which it was called.

The obvious problem with the scheme as outlined above is the prospect of unterminated left recursion. This may be eliminated using any standard grammar re-writing scheme (for example, that proposed in [3].) One approach is to incorporate a check for left-recursion into the **Match()** method of each class. For classes representing terminals or actions (**Token**, **Action**, etc.) this check is trivial. For classes representing rules, the check recursively traverses each component object, looking for and eliminating left-recursion wherever it occurs. When a given **Parser** object has been checked, it is marked as being non-left-recursive, allowing the check to be short-circuited on subsequent calls to **Match()**. Note that, using this approach, the assignment operator for Rule classes must be redefined to "unmark" a rule-type object (and all its component objects) when a new production is assigned to it.

6. Classes which represent actions

It is often desirable that a parser perform certain actions (such as caching data, computing interim results or constructing some data structure) at various points during a parse. It is relatively straight-forward to create another subclass of **Parser** (class **Action**) which manages such activities.

To be useful, actions must manipulate some form of data. To this end, a publicly accessible member (**Data**) is added to each parsing class. The **Data** member is a generic pointer which may be used to polymorphically access various kinds of values (see Figures 2 and 4 for examples of the use of the **Data** member within **Action** objects).

A simple approach to embedding actions is to assume that all actions to be performed during a parse are executed by calling some function, which takes a single argument representing the current context of the parse. Examples of such functions appear in Figure 2.

```
int  HelloWorld(Context  unused)
{
    cout << "hello world";
    return  1;
}

int  AddData(Context  context)
{
    context.parent->Data    =   new int;
    *context.parent->Data   =   *(int*)context.parent->Child(1)->Data   +
                                *(int*)context.parent->Child(3)->Data;
    return  1;
}

int  Fail(Context  unused)
{
    return  0;
}

int  Echo(Context  context)
{
    cout  <<  *(String*)context.action->Data  ;
}
```

Figure 2: Functions to perform parsing actions

An **Action** object contains a pointer (**embeddedfunc**) to a function of the above signature and has a constructor which enables the pointer to be initialized appropriately. **Action**s match an input stream by overloading their **Match()** method:

virtual Parser * Action::Match(Context)

...which executes the function to which **embeddedfunc** points and returns a pointer to the **Action** object if the embedded function returned non-zero (indicating success.) If the function returns zero (indicating failure), the method also returns zero.

Paralleling the manipulator concept used in C++ iostreams, the sequence and disjunction operators ("**+**" and "|") defined for the **Parser** class can be overloaded for arguments which are function pointers, thereby allowing action functions to be embedded directly into a grammar, without needing to be explicitly encapsulated in an **Action** object.

7. Resolving parsing ambiguities

Typically, in recursive descent parsers shift/reduce and reduce/reduce ambiguities are resolved by applying a universal selection scheme. For example, shift/reduce conflicts are normally resolved by favouring the shift and reduce/reduce conflicts by favouring the first applicable reduction. These make good defaults, but may not always represent the most appropriate behaviour for a dynamically lexed parser.

It is relatively straightforward to provide a significantly more flexible resolution mechanism, in which each **DisjunctionRule** object has its own disambiguating scheme. The **DisjunctionRule** class is given two extra members – a function pointer (**compare**) and a pointer to a **Parser** (**bestmatch**) – as well as a method (**MatchOn()**), which may be used to assign a value to the function pointer.

The function pointer is used to access functions which compare two matching **Parser** objects (the current "best" match and some alternative) and determine if the alternative represents a "better" match than the incumbent. Figure 3 illustrates four such functions. Each function returns non-zero if the alternative match is "better" and zero otherwise. Note that the function **Longest()** corresponds to the abovementioned default behaviour.

When a component **Parser** in a **DisjunctionRule** object succeeds in matching against the input stream, the function pointed to by **compare** is used to test the matching **Parser** against the previous best match (accessed via the **bestmatch** pointer.) If the function pointed to by **compare** returns non-zero, **bestmatch** is made to point to the new matching **Parser**, effectively supplanting the previous best.

Note that each of the comparison functions in Figure 3 deals with the special case of a null **bestmatch** pointer (which implies that the candidate is the first matching production) by returning non-zero (indicating the candidate is "better" than nothing.)

```
int Longest(Parser * current, Parser * candidate)
{
   if (!current) return 1;
   return candidate->NumberOfTokens() > current->NumberOfTokens();
}

int MostChars(Parser * current, Parser * candidate)
{
   if (!current) return 1;
   return candidate->NumberOfChars() > current->NumberOfChars();
}

int FirstMatch(Parser * current, Parser * candidate)
{
   return (!current);
}

int LastMatch(Parser * current, Parser * candidate)
{
   return 1;
}
```

Figure 3: Functions to compare matched productions

8. Conclusion and future work

The C++ programming language provides features such as constructors and operator overloading, which make it possible to "extend" the type system of the language in useful and largely transparent ways. This paper has outlined one such extension, which enables the programmer to incorporate a declarative style of parser declaration into standard C++ programs.

In addition, the technique permits the programmer to go beyond standard fixed look-ahead schemes by using (potentially context-sensitive) lazy tokenization and by specifying individual disambiguating schemes for each disjunctive rule within a grammar. Terminals and non-terminals may be polymorphically assigned values, which may in turn be accessed and operated on by user-defined actions embedded in the grammar.

Work is currently progressing on the creation of another set of classes, which will be used to represent "deferrable statements". The idea is to create objects which serve not as direct variables, but as proxies for components of the grammar rule in which they are embedded (like **$1**, **$$**, etc. in *yacc*.)

The standard arithmetic operators would be overloaded for these proxy objects in such a way that they would <u>not</u> perform the appropriate operation directly when executed (that is, during the construction of the grammar) but would instead create special **Action** objects. These objects would then perform the appropriate operation when their **Match()** method is called (that is, during parsing), effectively delaying the execution of actions specified within the grammar to that time.

References

[1] Hall, P.W. (IV), *Parsing with C++ Constructors*, ACM SIGPLAN Notices, vol. 28, no. 4, pp. 67-68, April 1993

[2] Johnson, S.C., *Yacc - Yet Another Compiler Compiler*, Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill NJ, 1975

[3] Aho, A.V., Sethi, R. & Ullman, J.D., *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, Massachusetts, 1986

```
#include "Parser.h"

int printval1(Context context)
{
      cout << *(int*)(context.parent->Child(1)->Data);
}

int sum1and3(Context context)
{
      int val1 = *(int*)context.parent->Child(1).Data;
      int val3 = *(int*)context.parent->Child(3).Data;
      context.parent->Data = new int(val1+val3);
}

int pass1(Context context)
{
      context.parent->Data   = context.parent->Child(1).Data;
}

int prod1and3(Context context)
{
      int val1 = *(int*)context.parent->Child(1).Data;
      int val3 = *(int*)context.parent->Child(3).Data;
      context.parent->Data = new int(val1*val3);
}

main()
{
      RegexToken   INT("[0-9]+");

      Token        PLUS("\+");
      Token        TIMES("\*");
      Token        EOL("\n");

      Rule         Expr, Sum Prod;

      Action       Print1(printval1);
      Action       Result1Plus3(sum1and3);
      Action       Result1(pass1);
      Action       Result1Times3(prod1and3);

      Expr    =    Sum + EOL + Print1
           |   Sum + EOL + Print1 + Expr
           ;
      Sum =    Prod + PLUS + Prod + Result1Plus3
           |   Sum + PLUS + Prod + Result1Plus3
           |   INT + Result1
           ;

      Prod    =    INT + TIMES + INT + Result1Times3
           |   Prod + TIMES + INT + Result1Times3
           ;

      Expr.Match(pstream(cin));
}
```

Figure 4: A simple expression parser