Parsing with C++ Constructors

Philip W. Hall IV
University of Cincinnati, MS student
hallpwcd@ucunix.san.uc.edu
hall@dec254
6733 Murray Ave.
Cincinnati, Ohio 45227

1. Introduction

This paper gives a brief sketch of how to build a recursive descent parser using C++ constructor functions.[1] The advantages gained from programming with C++ classes and constructors stem from the use of classes to organize the implementation of the parser and the fact that parser generation can be made a systematic translation of BNF-type production rules to class definitions.

2. Translation from rules to classes

The translation process from BNF-type context free grammar description to parser classes is straight forward. Recall from automata theory that a BNF grammar for a context free language has a set of production rules. Each rule has a left-hand non-terminal symbol, a delimiter, and a right-hand set of mixed terminal and non-terminal symbols. We also know that for each non-terminal symbol in the grammar there is at least one production rule in the grammar.

Each rule in the grammar is translated into a class definition, called a **grammar rule class**. An instance of a grammar rule class is called a **grammar rule object**. As with traditional recursive descent parsers, a function is created which attempts to consume the input associated with the rule in the grammar (Aho, 1986) Under the scheme introduced in this paper that function is a constructor for the grammar rule class. Each constructor function created in this way is passed a sentence object holding the sentence to be parsed. The sentence object must be capable of producing a token stream as a standard lexical analyzer does.

For each new grammar rule class, GRC, a constructor is coded so that it creates grammar rule objects for each non-terminal that appears in the right-hand side of GRC's production rule. The constructor is also coded to consume any terminal strings.

3. Parsing with Classes

An example will show a typical parse for a simplified grammar. First a key to notation:

Symbol	Description
<...>	non-terminal
|	signifies choice
"..."	signifies a terminal string
::=	separator

Example grammar:

```
<expr>    ::=   <digit><op><digit>
<op>      ::=   "+"|"-"
<digit>   ::=   "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

The translation process would produce three class definitions, one for each of the non-terminal symbols: <expr>, <op>, and <digit>. In the following code sample generated from the above grammar, identifiers which start with a capitalized letter denote class names and all lower case identifiers denote objects. Also, the class definitions for the objects "sentence" and "token" are omitted.

[1] see (Ellis, 1990) for a full treatment of C++ constructor functions.

```
Expr::Expr(Sentence &sentence)
{
  digit1 = new Digit(sentence);
  operator = new Op(sentence);
  digit2 = new Digit(sentence);
};

Op::Op(Sentence &sentence)
{
  sentence.get_next_token(token);
};

Digit::Digit(Sentence &sentence)
{
  sentence.get_next_token(token);
};
```

The following code shows how the parser might be invoked for this language:

```
Sentence *sentence;

sentence = new Sentence("1+2");
expr = new Expr(sentence);
```

Using C++ classes provides many convenient and intuitive ways to organize the implementation of a parser. For example, sentences are given the role of lexically analyzing themselves. Also, we might change the implementation of the grammar rule class private data so that it produces a list of grammar rule objects. This list would contain successful and unsuccessful parses. The reader, I'm sure, can imagine other additions and enhancements.

4. Discussion

The technique described in this paper is simple and of somewhat limited use. It is restricted to grammars which are parsable in a top-down fashion. However, its usefulness stems from two characteristics: it takes advantage of object-based program organization, making it easier to understand and modify, and it is fairly trivial to automate the process of generating parsers from BNF style grammar description. This technique is well suited as a tool for teaching recursive descent parsing.

My final example of a text-based message protocol will help illustrate why this technique is useful for simple context-free grammars.

Imagine an inter-process message is received as text string. A message protocol parser object is instantiated passing the text string to its constructor. When the application interprets the message it simply extracts commands and parameters from the message protocol.

This particular example suggests an enhancement to the parser: the ability to extract text strings from the parsed message object by the name (left-hand non-terminal symbol) of the grammar rule used to parse it. In this way, for example, the token text represented by a grammar rule called <source_process> (representing the process name of the sender of the message) could be extracted from the message protocol parser object by invoking a member function called get_value("source_process").

This enhancement has several advantages. It can be generalized for any grammar rule (of course repeated parses of the same rule would have to be disambiguated somehow). It would also be consistent with the BNF description given to specify the grammar, thus making the application conform more closely to the specification.

5. Conclusion

Using C++ class constructors to implement a recursive descent parser offers a variety of potential advantages over similar top-down parsing implementations, including encapsulation of grammar rules in objects, standardized interfaces to parsed representations, and a good environment for systematizing parser generation. With a few simple interfaces such as the get_value() function, a grammar rule object is made easy for an application to use as well as forcing it to conform more closely to the original grammar specification.

References

Ellis, 1990.
M.A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual,* Addison-Wesley, Reading, Massachusetts (1990).

Aho, 1986.
Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison Wesley, Reading, Massachusetts (1986). 0-201-10088-6