Parsers and Learning Programming in ProgrammingLand

Curt Hill

Mathematics and Computer Science Department

Valley City State University

101 College St SW

Valley City, ND 58072

701 845-7103

Curt_Hill@mail.vcsu.nodak.edu

Abstract:

The ProgrammingLand MOOseum is virtual environment devoted to the education of Computer Science students in their initial programming classes. It is an online, learner-centered museum of programming that promotes active learning. It serves a purpose similar to certain types of course management software, but is fully customizable in a way that most such software is not. The system provides content material, hosts interactive exercises, monitors the progress of students and gives assignments. It was designed for both distance education and the enhancement of classroom situations.

Table-driven parsers were introduced into the MOO to provide students with the ability to practice simple programming language constructions. Although every compiler contains a parser, compilers require a complete syntactic unit, such as a program, function or object as input. Yet in the early stages of learning a programming language, students may benefit from practice on simple constructions such as a declaration, an assignment statement, or a comparison, without running a compiler on an entire, albeit trivial, program. The MOO is not a general-purpose programming environment thus having a full compiler implemented within it is impractical. However, the table-driven parsers allow the students to practice what they have just learned in an easy manner and without leaving the MOO environment.

A new set of objects was created where a student could enter a single line of code and test the legality of the syntax. These objects contained parsing tables of the particular grammar subset in question. They also provide a simple lexical analyzer and parser for this grammar subset. The author obtained a program that produces tables for LR(1) languages and has developed a suite of programs that manipulates the grammars, converts the tables into needed format, as well as doing the parsing in a variety of contexts. This system has been used on subsets of the grammars of C++ and Java.

Introduction to ProgrammingLand.

The use of small parsers has been introduced into the ProgrammingLand MOO to give students an opportunity to exercise programming language syntax without the need to write a complete program. The ProgrammingLand MOO[Hill, 2001] was designed to become a vehicle for delivering programming instruction at a distance over the Internet. It has many things in common with a course management system, but a very narrow focus, programming and Computer Science. It has been used as a supplement to an introductory programming course during its development and has become essential to the course, since it has educational activities that no other component provides. The course is an introduction to programming in C++ for Computer Science students. ProgrammingLand has also been used as an experimental adjunct to a Java course, although that wing of the museum is much less complete.

The ProgrammingLand MOO provides many facilities to the course. A MOO [Curtis, 1992] is a derivative of the MUDs (Multiple User Domain), which themselves are derived from early games like Adventure and Zork. MOOs and MUDs provide text-based, virtual reality, where players explore rooms and interact with objects in the rooms and with other players. The virtual reality is structured as a series of rooms connected by exits with each other. The previous uses of MOOs and their predecessors MUDs was the creation of text-based fantasy or social worlds where users could interact with one another and the contents of the world. Although a MOO is a text-based reality, they have been adapted to be accessible by web browsers.

ProgrammingLand is not made to resemble a castle or dungeon inhabited by dragons and other creatures of fantasy. Instead it has a metaphor of a museum of programming. Students move from room to room and view the information presented in way analogous to that of browsing a series of web pages. The principle virtue of a MOO over a course management system (or simple series of web pages) is the ease of customization over these types of systems. The script language has similarities to C++, but is object oriented in a different way. Each object, such as a room, player, exit or interactive device is an object that can be easily modified. Hence the system may be changed in very fundamental ways. Thus, most of what follows can only describe ProgrammingLand and not any other MOO.

ProgrammingLand organizes groups of rooms into lessons, where a lesson contains a single topic. Each of these has sets of requirements for completing the topic. These requirements may include the viewing of one or more rooms of material, the execution of interactive exercises and completion of previous lessons. When a student leaves a lesson the system checks their progress towards completion. If the lesson requirements are satisfied then the MOO announces to the student that they have completed the lesson. Otherwise they are shown the requirements that are not yet fulfilled.

ProgrammingLand may also make assignments using the lesson completion mechanism. A lesson may be as small as a few rooms to as large as the entire course. Completion of certain lessons signals readiness to begin a programming assignment. When a student completes one of these lessons an agent visits the student and gives them the description of the assignment. The happy side effect of this scheme is that unlike a textbook, the students must use ProgrammingLand or they do not get assignments and cannot receive the scores needed to do well in the course.

The use of interactive exercises, such as simulations, is an important asset to ProgrammingLand. The MOO integrates the interactive exercises and may require them in the lesson structure. A Java applet in a web page should produce the same experience in a student. The problem with such applets is how (or how often) it is used. Although it is technically possible for data gleaned by the applet to be captured by the server, it is seldom done in practice. Yet, ProgrammingLand has several kinds of interactive exercises that integrate seamlessly into the entire system.

The main interactive object in ProgrammingLand is the code machine. This object contains a short program or fragment of programming code. The student may display the code, ask for a line-by-line explanation of what the code does or ask for a simulated execution. Either the explanation or execution may be used as a requirement for a lesson. Although the MOO has an object-oriented scripting language that resembles C++, performance considerations prevent the server from doing compiles or executions of student programs. Thus a simulation is an effective way to give the student valuable experience without overloading the server. The current server for ProgrammingLand is 700 MHz Pentium running FreeBSD.

One of the advantages of ProgrammingLand over course management systems such as Blackboard [Yaskin, 2001] or WebCT [WebCT, 2001] is the ease of customizing the environment to better fulfill the goals of the project. The Programming Languages class is often assigned the task of developing new interactive objects that may be of some use [Hill & Slator, 2000]. Few such class projects have been suitably robust and useful, but this illustrates the ease of enhancing the MOO. The hard part is not the implementation, but the original idea of something that will be interesting and educational. A previous object, the workbench, had attempted to allow student specification of a construction that would be parsed. However, it was too difficult for the student to describe the construction of the statement to be of much use.

Parsing Objects.

The inspiration for the new objects was an exercise[Horstmann, 2003b] that accompanied a textbook[Horstmann, 2003a]. In it a student was presented with a line of code. They were to say whether the code was syntactically correct or not. If it was in error they were to provide a correction. A fragment of the lab exercise is shown below:

Are the following ints and doubles properly declared and/or initialized? If not, supply a correction.

```
     int 122;
     double pi_times_10  314.159;
     double = 314.00
```

This exercise was done on paper in the introductory Java class at NDSU. This inspiration resulted in two objects, a practice object and syntax test object, that are being introduced into ProgrammingLand. The practice object contains all the parser functionality. The syntax test is its descendent and embeds this into a true/false with corrections test, where the parser judges the corrections.

A practice object asks the student to type in a single line of code. It then parses this code based on the parse tables that are properties of this object. Since it is a parse and not a compile it merely pronounces whether the offered line of code is correct or not. The room containing the practice object indicates the construction that it will look for, although that is usually clear from the surrounding exhibits.

A syntax test object, like a practice object, is usually placed in a workroom, since a workroom is one of the few rooms that permits only a single student at any one time. The test may only be taken once, so the first thing the object does is to ask whether they are ready to take the test or not. It is a good policy to attach a room with a practice object every room that contains a syntax test object and the two would use the same parse tables. In this way a student may practice the indicated syntax without a recorded evaluation. The syntax test maintains a list of students that have already taken the test to prevent a student repeating the test.

The syntax test has a group of right and wrong statements as well as some counts as to how many may be offered. Typically a test will be instructed to offer two correct and four incorrect statements. These are randomly selected from larger lists of correct and incorrect statements and then shuffled so that students will not usually receive a test identical to that of a friend. A single statement is presented to the student and then the student is prompted for a yes or no answer to the question of is the statement correct. If they are correct that the statement is legal they move on to the next question. If they are correct that the statement is not legal they are prompted for a correction. This correction is scanned and parsed by the object and the test moves on to the next question. At the end of the test the object number of the test and the count of right and wrong responses is written onto the student's object. A lesson may then use the presence or absence of this event in its requirements. The practice object also posts an event to the student object but this is for study of the patterns of usage and may not be used in the requirements of a lesson.

An annotated transcript of a usage of a syntax test is shown. The user input is in **bold** and prefixed by "==>" for clarity. Explanatory comments are inserted on lines in italics. This will show only three of the six tests presented.

```
Declaration test
    This workroom contains a Syntax_Test object named
Declare, which has an alias of D. One of the requirements
of the lesson is to take this test. In it you will be
shown six declarations. You will be asked if they are
right or wrong. If you say it is incorrect (and it is)
then you will be given a chance to give a correct
statement.
    Since you only get to take the test once you are
encouraged to use the practice object that is next door
to hone your skills.
    If this test does not automatically start, you may
begin by typing:
 test declare
You see declare here.
Obvious exits: [exit] to Declaring Primitives, [practice]
to Declaration Practice Workroom
```

[This workroom executes the test automatically. The text above is the room description, the text below is the interaction of the syntax test object and user. Notice that the Declaration Practice Workroom contains the practice object that accepts the same grammar as this test object.
Before the following message is displayed, the object has determined that the user has not yet taken this test.]

```
You may only take this test once, are you ready to take
it now? [Enter `yes' or `no']
==>yes
```

[The general instructions:]

```
I will now show you a series of 6 constructions and you
will tell me if they are legal or not. If they are not
valid then I will ask you for a correction, which I will
scan for validity.
You will not be able to do any other commands until the
test is over.
```

[The first of the six:]

```
Consider the following:
int 144;
Is this correct? [Enter `yes' or `no']
==>n
```

[Yes and no may be abbreviated to just one letter or spelled out.]

```
Yes. This statement is not correct. Now you will enter
the correction.
[Type Enter the correction. or `@abort' to abort the
command.]
==>int a = 144;
Input line:     int a = 144;
Checking...
That is correct.
```

[The test continues, with several interactions omitted:]

```
Consider the following:
double e_times_10 = 271;
Is this correct? [Enter `yes' or `no']
==>yes
That is right
Consider the following:
int value = 5
Is this correct? [Enter `yes' or `no']
==>yes
No. This statement is incorrect.
```

[The end of the test, also with lines omitted:]

```
That concludes the test. You answered 4 out of 6
correctly.
```

The student exits the workroom when they complete the test and go about their learning. It should also be observed that the syntax test is completely handled by the MOO. There is no paper to hand in, no person needs to correct the lab, and no person needs to check if any student has fulfilled the requirements.

The description of how a parsing object is constructed and how they were implemented is considered in the following sections.

Construction of Parsing Objects.

The first step in constructing parser objects for the MOO is to obtain a grammar for the language in question. The ProgrammingLand museum has important wings on C++ and Java, both of which have published grammars. A typical lesson only considers a small portion of the syntax of a programming language, such as a declaration, an assignment statement, or the header of a decision. In such cases the grammar could be constructed by the instructor, but this increases the likelihood of error. The full grammar is much too large for this type of parser or the MOO, so the next step is to reduce the full grammar to just the intended construct. This requires some discretion as to what is needed and what is not. For example the first coverage of simple types in C++ does not need any reference to arrays or pointers, although the length modifiers such as short and long may or may not be retained.

This process is iterative, since the goal is to have an unambiguous and useful grammar, with tables of a reasonable size. The usual way to find the table size is process the grammar with the parser generator in question. There are many generators available such as YACC, Bison[GNU, 2003], JavaCC[WebGain, 2003], or CUP[Hudson, 1999]. Most of these are compiler-compilers, hence their goal is to produce a compiler, which has some advantages and disadvantages. The compiler-compilers generally make it easy to attach the semantic routines and their output is often a program, which should be the basis of a compiler. In this situation a compiler is not the desired result and a C or Java program is even less desirable. Therefore, what was used was a very old parser generator named LR. This is a FORTRAN program written by Alfred W. Shannon [Wetherell & Shannon, 1981] for Lawrence Livermore Laboratory under government support. It produced as output several tables of integers. These tables are then used as input to the parser object.

Two table driven parsers were developed. The practice object was in the MOO. However, testing the grammar and parser required something more convenient than loading all the data into ProgrammingLand and then exercising the practice object. Therefore a stand-alone table parser was developed in C++ to run under the Windows platform. The latter loads in the tables produced by LR to initialize the parser. It will parse either a single line typed in a dialog box or a specified file. The parsing and lexical analysis is similar to that in the practice object. A Java applet could use a similar approach. Another program was developed that fits in between LR and the two table parsers. It merely scans the LR output and produces input suitable for the MOO or the C++ table parser.

Once the grammar has been trimmed to only exercise the desired features and has been tested, then the MOO construction commences. The needed rooms are built and a practice object is placed in one and the syntax test object in the other. Both objects take the same set of tables, but a syntax test has four additional properties that must be set. There are two lists of constructions, one of correct constructions and the other incorrect. There are also a couple of integers or how may correct and incorrect questions to present to the student. In the example given above the test was to give two correct and four incorrect statements. The statements are randomly selected from the lists and then randomly ordered by the test object for presentation to the student.

Implementation of the Practice and Syntax Test Objects.

The LR parser generator came with a sample parser using the generated tables. This parser was approximately 500 lines of FORTRAN, with many redundant common declarations, because of FORTRAN's lack of global declarations. The source code was made up of about ten functions, but did not include a lexical analyzer. Therefore the practice object and the grammar-testing program needed these ten routines converted into the appropriate programming language (Lambda MOO or C++). This was not as hard as it might sound, since the parser tables and all the internal data are always arrays of integers. Only the lexical analyzer needs to deal with character data. Although FORTRAN may have unusual flow of control constructs, most of the spaghetti code that gave the language a bad name was absent. Shannon had mostly used rather conventional flow of control constructs, even though he made them from one line IFs and GOTOs. Thus conversion to C++ or Lambda MOO code was relatively easy.

The lexical analyzer was somewhat more interesting. It needed to be designed in a way that allowed it to not know what reserved words or other terminals it had until run-time. However, the range of languages that it would reasonably be expected to parse have much in common. There are six special tokens that it must know something about: identifiers, integer constant, floating constant, string constant (enclosed in "double quotes"), character constants (enclosed in 'apostrophes') and the end of source. C++ and Java use both types of quoted strings, while Pascal only uses the latter. However, the lexical analyzer makes no attempt to see if the usage is correct, such as a character constant having multiple non-escaped characters. This is illegal in C++ but correct in Pascal. It further assumed that a blank was a delimiter, although no blank would be needed between an operator and variable or reserved word; identifiers and reserved words could include letters, digits and underscores, though they must start with a letter; multiple operators could begin with the same character; and similar rules. It then reads in groups of characters that it thinks could be tokens and categorizes them by initial character. It then looks for the longest initial sub-string that is a valid token, removes that from the string and keeps processing. Hence the C++ expression:

```
c=a+++c;
```

will be processed as:

```
c = a ++ + c ;
```

This may or may not be sufficient for a compiler, but for this application it is a useful approximation.

Records and administration.

A lesson may require that a student take a syntax test, though not a practice, to fulfill the requirements. However the requirements do not look at the scores of this test. A student may answer all the questions incorrectly and still fulfill the requirements. Although the MOO does extensive record keeping as to what the student has done, there has not yet been an implementation of anything like a grade book. Instead, the MOO is periodically scanned by an agent, which transmits the data to a SQL compliant database [Hill, 1999]. This database may then be queried for a variety of actions and this data exported to a spreadsheet or any other suitable grade book. This simplifies what the test objects need to record on each player object.

Each syntax test object records the object numbers of the students who have taken the test. This may be shown by a MOO command to a player of sufficient authority, but not the student. Of course, it is also is present in the SQL database. When the student completes the test, two events are recorded on the student object. The first is that they have completed the test, which is what the lesson requirements look for. The second is the score of their test, the number of questions that were right and the number that were wrong. Currently the only mechanism for observing this data is through the SQL database or for a MOO wizard character to manually examine the events property.

Problems and future work.

One of the goals of this effort was to provide a new object type that had several desirable characteristics. The object should provide a good educational experience, be easy to produce, the processing needed for the object would be well within the server's capabilities and subsequent objects would require no additional MOO programming.

Determining whether this educational experience is good or not is rather complicated, however there is one glaring flaw with either the practice object or syntax object, the error recovery is inadequate. If the parser determines that the construction is incorrect the student gets poor feedback as to what was incorrect. The best it can do is signal where the error was found.

A shift-reduce parser typically uses little semantic information. In a compiler, a reduction triggers an action, such as storing a variable name into a symbol table or checking the types of the two sides of an operator. The presence of a symbol table would require more processing than is acceptable and would also require customization of each object. Hence complete type checking is out of the reach of these objects. Two techniques were used to give minimal semantic checking.

The LR supplied parser calls a routine named Synth and passes it the production number where the right hand side is about to be reduced to the left hand side. Simple semantic checking may be done in this routine. In the MOO the presence of a function may be determined before it is called. Thus the parsing routines of the practice object do not provide this routine, but check to see if it is present and only call it if exists on this object. If additional semantic information is needed a test object may implement the Synth routine, but it may also remain unimplemented without problems. In the case of a Java primitive declaration it looks for the reduction of one of the constants into an expression and captures the type of the constant, if present. It also captures the reduction of any primitive type into a primitive type non-terminal. When the reduction of an assignment is found the two captured types may be checked.

Another ad hoc solution was devised which attempts to solve a related problem. Consider the following Java declarations:

```
int value;
final int Max = 10;
double x = 1.5;
```

A grammar that handles declarations should accept all three. However a test that is looking for a constant declaration could use the same grammar. A simple semantic check is to require that a successful line needs certain tokens to be present. Thus a test that is looking for constant declaration may use this grammar, but demand that the **`final`** token and the **`=`** token be present in the line. The implementation of this is that a parser object has a Needs property, which contains all the tokens that are needed, beyond the minimum. Should one of these required tokens not be in the string an error message may show that this is missing.

Neither the needed token technique nor the monitoring of reduction technique can find all possible problems. A syntax test has only a single grammar. Thus if the test asks for the correctness of:

```
    double d 5.0;
```

the student could give as a correction:

```
    int i = 5;
```

or even

```
    int i;
```

without the system detecting an error. Specifying the assignment operator as a needed token prevents the latter, but unless the test only considers double declarations it cannot prevent the integer declaration. Usually the students do not know this, they assume the system is more intelligent than it actually is.

There is a danger a student will conclude that an illegal construction is actually valid. However, such constructions should be comparatively rare. This is really the same problem as a compiler that is not compliant with the standard language, except the parser objects are used substantially less than the compiler. The parser and syntax test objects do not put an undue burden on the server, such as full checking would.

The parser objects also tend to be large in comparison with ordinary objects. The average object is about 1K in size, a room object is typically about 1-2K bytes, a student object about 4K, but a syntax test is around 10-12K. This bulk is from the parser tables that are part of the object. The grammar for Java declarations, which includes constants, primitives and initialization, contains 53 tokens (terminals and non-terminals) and 52 productions. The tables, except the vocabulary contains 783 integer entries. (This grammar is shown in the appendix.) Currently both the terminal and non-terminal names are stored on the object. The terminal tokens are needed by the lexical analyzer. The non-terminals are not yet used, but are stored for future enhancements to error processing.

The current ProgrammingLand database is about 12M in size so having a small number (eg. 50) of parse objects is not going to greatly affect the size. If the size of these objects does become an issue a potential solution is sharing tables. A syntax test usually has a practice object nearby containing an identical grammar. An improvement that has not yet been implemented is having a third object that stores the tables for both the test and practice objects. This could reduce the space requirements without any affect on performance.

The parser objects were tested on a introductory Java class, however errors in the setup of the experiment prevent any strong conclusions. These errors include the ability of students to withdraw from the experiment at any time and a lack of a suitable control group. Despite these problems the results were positive. Correlating the number of times a test or practice object was executed against exam scores produced a correlation coefficient of 0.39, which is significantly positive. (This correlation only included students who did take the exam, for in an introductory, freshman class there are considerable number of students who drop the class.) This positive correlation could be explained by the hypothesis that the parse objects helped students but it could be equally well explained by the hypothesis that the good students were more likely to exercise a parse object than the poor students.

Conclusions.

The implementation and first tests of the parser objects in ProgrammingLand went as well as could be expected. The parser table generation software works adequately; the programs to convert the tables into MOO properties is automated; the practice and syntax test objects perform well; and the MOO can now accept the usage of the test objects as a requirement. The objects were introduced at a time when only a Java class was available to test them. The fall semester will allow a wider distribution of the objects into the C++ part of the MOO, which is better developed, at least from an instructional view. Thus from a technical point of view the project has been successful though more work is needed.

The educational value of these items is yet to be determined and must await a more precisely constructed experiment. The important question of the usefulness of such objects must await a later term.

References.

Curtis, Pavel (1992). Mudding: Social Phenomena in Text-Based Virtual Realities. Proceedings of the conference on Directions and Implications of Advanced Computing (sponsored by Computer Professionals for Social Responsibility)

GNU (2003). Bison – Replacement for the parser generator 'yacc'. http://www.gnu.org/directory/GNU/bison.html Date accessed 28 February 2003

Hill, Curt (2001). Evolution of ProgrammingLand. Proceedings of the Midwest Instructional and Computing Symposium (MICS 2001), April 2001, Cedar Falls IA.

Hill, Curt and Brian M. Slator (2000). Computer Science Instruction in a Virtual World. World Conference on Educational Media, Hypermedia and Telecommunications (ED-MEDIA 2000), June 26-July 1, 2000, at Montreal, Quebec, Canada.

Hill, Curt (1999). Extracting Data from a MOO. Proceedings of the Small College Computing Symposium, 1999. April 1999, LaCrosse WI.

Horstmann, Cay (2003). *Computing Concepts with Java Essentials*, 3rd ed. New York, John Wiley & Sons.

Horstmann, Cay (2003). Laboratory Notebook, Chapter 2 – Fundamental Data Types. http://www.horstmann.com/ccj2/labs/ccj2_ch2.html. Date accessed 28 February 2003. Date accessed 3 March 2003.

Hudson, Scott E (1999). CUP, LALR Parser Generator for Java. http://www.cs.princeton.edu/~appel/modern/java/CUP/. Date accessed 28 February 2003.

WebCT (2002). WebCT Software & Services. http://www.webct.com/products Date accessed 5 March 2002.

WebGAIN (2003). Java Compiler Compiler (JavaCC) – The Java Parser Generator. http://www.webgain.com/products/java_cc Date accessed 28 February 2003.

Wetherell, Charles & Alfred Shannon (1981). LR- Automatic Parser Generator and LR(1) Parser. IEEE Transactions on Software Engineering, vol. 7, no. 3, May 1981, pp. 274-278.

Yaskin, David, Stephen Gilfus (2001). Blackboard 5, Introducing the Blackboard 5: Learning System. http://company.blackboard.com/docs/cp/orientation/EnterpriseLearningWhitePaper.pdf. Date accessed 5 March 2002.

Appendix – A sample gramar

```
 1 <SYSTEM GOAL SYMBOL> ::= $END$ FieldDeclaration $END$

 2 FieldDeclaration ::= FieldVariableDeclaration ;

 3 FieldVariableDeclaration ::= final PrimitiveType VariableDeclarators

 4 PrimitiveType ::= boolean
 5                / char
 6                / byte
 7                / short
 8                / int
 9                / long
10                / float
11                / double
12                / void

13 VariableDeclarators ::= VariableDeclarator
14                      / VariableDeclarators , VariableDeclarator

15 VariableDeclarator ::= DeclaratorName = VariableInitializer

16 VariableInitializer ::= Expression

17 DeclaratorName ::= IDENTIFIER

18 ComplexPrimary ::= ( Expression )
19                 / ComplexPrimaryNoParenthesis

20 ComplexPrimaryNoParenthesis ::= IDENTIFIER
21                              / Literal
22                              / BoolLit

23 PostfixExpression ::= ComplexPrimary
24                    / RealPostfixExpression

25 RealPostfixExpression ::= PostfixExpression ++
26                        / PostfixExpression --

27 UnaryExpression ::= PostfixExpression
28                  / ++ UnaryExpression
29                  / -- UnaryExpression
30                  / ArithmeticUnaryOperator CastExpression

31 ArithmeticUnaryOperator ::= +
32                          / -

33 PrimitiveTypeExpression ::= PrimitiveType

34 MultiplicativeExpression ::= CastExpression
35                           / MultiplicativeExpression * CastExpression
36                           / MultiplicativeExpression / CastExpression
37                           / MultiplicativeExpression % CastExpression

38 AdditiveExpression ::= MultiplicativeExpression
39                     / AdditiveExpression + MultiplicativeExpression
40                     / AdditiveExpression - MultiplicativeExpression

41 CastExpression ::= UnaryExpression
```

```
42                  / ( PrimitiveTypeExpression ) PostfixExpression

43 AssignmentExpression ::= AdditiveExpression
44                        / IDENTIFIER AssignmentOperator Expression

45 Literal ::= CHAR-LITERAL
46           / STRING-LITERAL
47           / INT-LITERAL
48           / FLOAT-LITERAL

49 BoolLit ::= true
50           / false

51 AssignmentOperator ::= =

52 Expression ::= AssignmentExpression
```