Homosapien Modeling Language (HML)

Final Report

COMS W4115: Programming Languages and Translators

Professor Stephen A. Edwards

Computer Science Department

Summer 2008 Columbia University

Dated: 2008-08-10

Derek Ng
dn2150@columbia.edu
Columbia Video Network

Contents

Introduction

Background

For my course project, I attempted to design my own programming language, the Homosapien Modeling Language (HML). I chose to tackle a language that would help programmers model our environment, by first allowing them the ability to define entities (i.e. humans) and then external forces that could affect these entities.

Goal

By allowing programmers to define individual people and how they are affected by the world around them, they might be able to realize the impact on larger societies. For example, the impact of a failing economy might force people to become less happy and this feeling might influence others they interact with. If written correctly, then HML should provide a data repository for analysts to analysts to extrapolate trends.

Main language features

HML is in a very infantile state. While it posses the skeleton to perform limited mathematical operations with primitive coding, it demonstrates that the language can recognize source code, interpret it, and act upon it. With more advanced developer support, the language could be extended to meet its visionary goals.

Tutorial

For Windows based environments, one must first compile the HML code with Java and ANTLR via the command:

```
>dir
\project\src
08/10/2008  09:16 PM    <DIR>          .
08/10/2008  09:16 PM    <DIR>          ..
08/10/2008  08:59 PM             2,721 hml_grammar.g
08/10/2008  09:16 PM               786 Main.java
               2 File(s)          3,507 bytes

>java antlr.Tool hml_grammar.g
ANTLR Parser Generator   Version 2.7.7 (20060930)   1989-2005
```

The ANTLR tool generates Java code based on my HML specifications.

```
>dir
\project\src
08/10/2008  09:16 PM    <DIR>          .
08/10/2008  09:16 PM    <DIR>          ..
08/10/2008  09:16 PM            16,447 HmlLexer.java
08/10/2008  09:16 PM             2,778 HmlLexer.smap
08/10/2008  09:16 PM            11,962 HmlParser.java
08/10/2008  09:16 PM             2,493 HmlParser.smap
08/10/2008  09:16 PM               510 HmlParserTokenTypes.java
08/10/2008  09:16 PM               320 HmlParserTokenTypes.txt
08/10/2008  09:16 PM             4,054 HmlTreeWalker.java
08/10/2008  09:16 PM             1,202 HmlTreeWalker.smap
08/10/2008  08:59 PM             2,721 hml_grammar.g
08/10/2008  09:16 PM               786 Main.java
              10 File(s)         43,273 bytes
```

At this point, one must compile the Java code.

```
>javac *.java
Note: HmlLexer.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

>dir
\project\src
08/10/2008  09:16 PM    <DIR>          .
08/10/2008  09:16 PM    <DIR>          ..
08/10/2008  09:16 PM             8,580 HmlLexer.class
08/10/2008  09:16 PM            16,447 HmlLexer.java
08/10/2008  09:16 PM             2,778 HmlLexer.smap
08/10/2008  09:16 PM             6,629 HmlParser.class
08/10/2008  09:16 PM            11,962 HmlParser.java
08/10/2008  09:16 PM             2,493 HmlParser.smap
08/10/2008  09:16 PM               773 HmlParserTokenTypes.class
08/10/2008  09:16 PM               510 HmlParserTokenTypes.java
08/10/2008  09:16 PM               320 HmlParserTokenTypes.txt
08/10/2008  09:16 PM             2,691 HmlTreeWalker.class
08/10/2008  09:16 PM             4,054 HmlTreeWalker.java
08/10/2008  09:16 PM             1,202 HmlTreeWalker.smap
08/10/2008  08:59 PM             2,721 hml_grammar.g
08/10/2008  09:16 PM             1,388 Main.class
08/10/2008  09:16 PM               786 Main.java
              15 File(s)         63,334 bytes
```

The warnings are a result of using the string “or” and “and” in the parser. They are nothing to worry about. At this point, to run any HML code, one just types “java Main < source_code.txt”.

Language Reference Manual

Lexical Conventions

Comments

Comments are preceded and followed by a control sequence. Comments are started with a forward, double slash “//” and are ended with a backward double slash “\\”. An example comment would be as follows:

```
        // THIS IS A COMMENT \\
```

Separators

Tokens are separated by blanks, tabs, and newlines.

Identifiers

An identifier in HML is a consecutive pattern of letters (a to z and/or A to Z).

Keywords

The logic operators “or” and “and” are the only reserved words in this language.

Numbers

Numbers are consecutive pattern of digits (0 to 9). Numbers are restricted to integers and may be positive or negative.

Other tokens

HML has a set of built in operators to manipulate variables.

```
(       )       +       -       *       /       %       ^
```

Types

Due to the restricted set of supported operations, HML is restricted to manipulating integers. But due to the nature of some of the mathematical operations, the default resulting data type is a double.

Expressions

Or expressions

The “or” operation in the term “A or B” performs the logical OR operation between A and B. At this time, the lexer recognizes this term but the OR operation has not been programmed.

And expressions

The “and” operation in the term “A and B” performs the logical AND operation between A and B. At this time, the lexer recognizes this term but the AND operation has not been programmed.

Arithmetic expressions

The “+” operation in the term “A + B” adds the integer A with the integer B and returns the sum as a double.

Subtraction expressions

The “– “operation in the term “A – B” subtracts the integer value of B from integer A and returns the difference as a double.

Multiplication expressions

The “* “operation in the term “A * B” multiples the integer value A with the integer B and returns the product as a double.

Division expressions

The “/ “operation in the term “A / B” divides the integer value B from the integer value A and returns the resultant as a double. At this time, the compiler confuses this operation with the comment delineators.

Modulo expressions

The “%“operation in the term “A % B” performs the modulus operation on integer A with the integer modulo B and returns the resultant as a double.

Exponential expressions

The “^“operation in the term “A ^ B” raises the integer A to the integer B power and returns the resultant as a double.

Operator precedence

HML uses the operator precedence denoted in the table. Operators in the upper rows take higher priority than operators in the lower rows.

Higher Priority	or | and
	+ | -

	* | / | %
Lower Priority	^

Expression

HML requires all expressions to be enclosed with parenthesizes and then end with a semicolon. For example, the following is a valid expression "(A + B);" while "A + B;" is not due to the missing parenthesizes. Based on external guidance and personal experimentation, the semicolon helps to distinguish between sequential expressions. Right now, the parenthesizes are included to define the boundaries of a single expression. It was planned to support parenthesizes to allow more complex, nested expressions where the parenthesizes would allow the programmer to manipulate the traditional operator precedence.

Project Plan

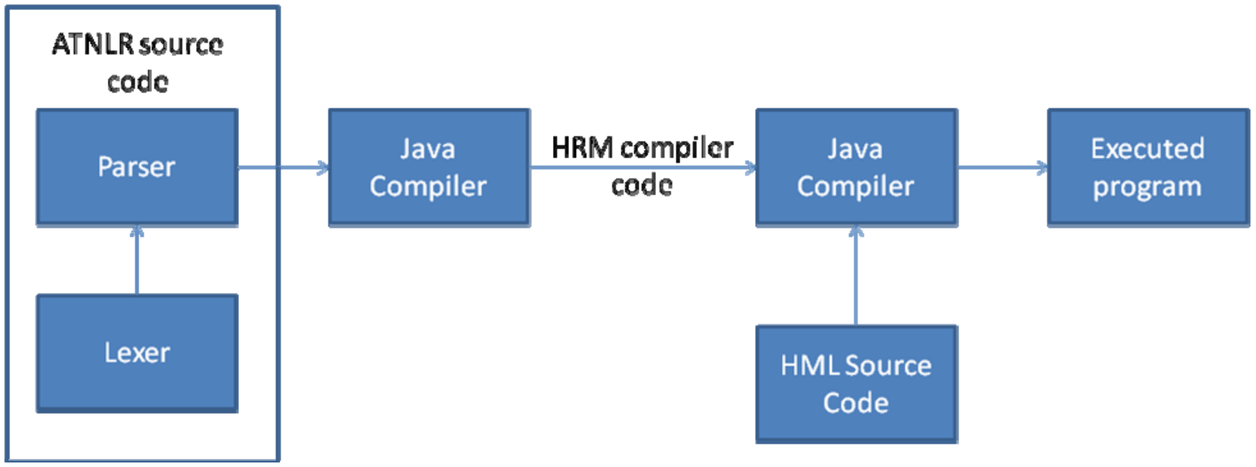
As a 1 man team, I was responsible for all aspects of the assignment from documentation to coding to testing. I submitted the original proposal, Language Reference Manual, and this final report while attempting to fold in feedback from Professor Edwards. For development, I researched ANTLR and wrote the lexer, parser, and AST walker. Base on my abilities and the roles expressed in lecture, I believe I would have been most effective doing documentation and/or testing.

Almost all of the semester was spent researching the role of each component in the compiler by reading numerous online Frequently Asked Questions (FAQs), holding electronic exchanges with other ANTLR developers, and repetitive trial and error coding.

The original plan was to research until the midterm, transition to writing out the compiler until the final, and then spending the last week documenting my findings. With the difficulty of the coursework itself and personal difficulties with learning the language, I ended up spending all my time until the final learning and trying to write ANTLR code with the last week spent documenting.

Version control was rather simple since I was the only person allowed to modify the code. Therefore I uploaded my code to my gmail e-mail account for offline storage at the end of the day when I modified the code.

Architecture Design



The above diagram represents the workflow my HML language. I generate the ANTLR code which gets passed to the Java compiler which builds my HML compiler in Java source code. The HML compiler source code then runs through the Java compiler with HML source code as an argument to run a HML program.

Currently there is a bug in the HML ANTLR code where the lexer will tokenize all of the source code, the parser will analyze all the tokens, but when will only evaluate the first line of the source code when passing through the tree walker.

Testing Plan

Multiple HML codes were generated to test individual features of the compiler. In each case, the AST was generated and displayed to screen. The first line returned from the compiler is the parse treem.

Test Case 1 – addition

```
>java Main < 01_add.txt
 ( ( ( + 1 2 ) ) null
Value: 3.0
```

Test Case 2 – subtraction

```
>java Main < 02_sub.txt
 ( ( ( - 10 2 ) ) null
Value: 8.0
```

Test Case 3 – multiplication

```
>java Main < 03_mult.txt
 ( ( ( * 300 4000 ) ) null
Value: 1200000.0
```

Test Case 4 – division

```
>java Main < 04_div.txt
Exception: line 1:7: expecting '\n', found '<EOF>'
```

As mentioned earlier, the current division operation confuses the division operator with the comment delineator.

Test Case 5 – modulus

```
>java Main < 05_mod.txt
 ( ( ( % 9 4 ) ) null
Value: 1.0
```

Test Case 6 – exponentiation

```
>java Main < 06_exp.txt
 ( ( ( ^ 13 45 ) ) null
Value: 1.3410681671324994E50
```

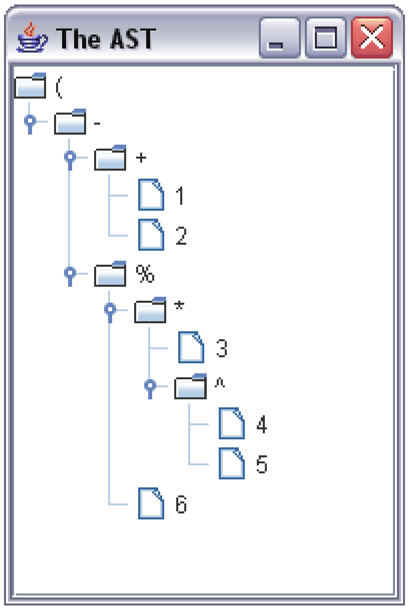
Test Case 7 – comments

```
>java Main < 07_comments.txt
Ignoring comment: //THIS IS A COMMENT\\
 ( ( ( + 3 4 ) ) null
Value: 7.0
```

Test Case 8 – multiple operations

```
>java Main < 08_combo.txt
 ( ( ( - ( + 1 2 ) ( % ( * 3 ( ^ 4 5 ) ) 6 ) ) ) null
Value: 3.0
```

The AST in displayed to visualize how it is generated.



Test Case 9 – multiple lines of code

```
>java Main < 09_combo.txt
 ( ( ( - ( + 1 2 ) ( % ( * 3 ( ^ 4 5 ) ) 6 ) ) ) ( ( ( - ( + 1 2 ) ( % ( * 3 ( ^ 4 5 ) ) 6 ) ) ) null
Value: 3.0
```

As mentioned earlier, the compiler will tokenize and parse the whole source file, but only evaluates the first line. The correct response would return 2 "Value: 3.0".

Lesson Learned

This exercise has taught me many things:

1. The biggest roadblock for programming HML was the difficulty I faced with learning ANTLR syntax. I personally require face to face interaction to learn foreign syntax. Since I am a remote student, I attempted this semester to rely exclusively on electronic exchanges by reaching out to other developers on the Internet to understand ANTLR. With software development being so fast paced, most developers I contacted had used ANTLR 2.0 "a while ago" and were a bit grainy on providing me advice. I was able to develop my own primitive lexer, parser, and AST walker which does show I was able to learn the basics of ANTLR.

2. My second biggest roadblock was that I had underestimated the dependence on Java in this class. Coming from a C background, I had been always able to see the parallels between C and Java to accomplish the assignment, but I feel my lack of experience with Java played more of a hindrance in this assignment than in others.

3. As a consequence of the first two difficulties, I believe one of the most obvious lessons I learned was that I hoped for more than I was capable of doing. Although minimal for some, I believe customizing my language to read in numbers, strings, "simple" mathematic operations, and comments were big accomplishments for me. It shows that I was able to walk through all the stages of a compiler.

Code listing

```
\project\src
08/10/2008  09:16 PM    <DIR>          .
08/10/2008  09:16 PM    <DIR>          ..
08/10/2008  08:59 PM             2,721 hml_grammar.g
08/10/2008  09:16 PM               786 Main.java
               2 File(s)          3,507 bytes
```

Credits

ANTLR is a difficult language for me to learn and without physical interaction with other classmates to ask for guidance, I had to rely exclusively on web based support. I would like to thank the following for taking the time to write their support pages.

- Daniel Ostermeier and Jason Sankey (of alittlemadness.com) gave me much insight with their ANTLR By Example webpage.

- Ashley Mills (of The University of Birmingham) for her clear ANTLR tutorial.

- Most importantly, Scott Stanchfield (of javadude.com) for writing up An ANTLR 2.0 Tutorial and responding to my e-mail questions.

Source code

hml_grammar.g

```
// **************************************************************************//
// Parser //
// **************************************************************************//

class HmlParser extends Parser;

options {
  k = 2;
  buildAST = true;
}

imaginaryTokenDefinitions :
   SIGN_MINUS
   SIGN_PLUS
;

program  : ( statement )* EOF;
statement: log_expr SEMI!;
log_expr : expr ("or"^ | "and"^ expr)*;
expr     : LPAREN^ sumExpr RPAREN!;
sumExpr  : prodExpr ((PLUS^|MINUS^) prodExpr)* ;
prodExpr : powExpr ((MUL^|DIV^|MOD^) powExpr)* ;
powExpr  : signExpr (POW^ signExpr)? ;
signExpr : (
            m:MINUS^ {#m.setType(SIGN_MINUS);}
          | p:PLUS^  {#p.setType(SIGN_PLUS);}
          )? atom ;
atom     : INTLIT;

// **********************************************************************//
// Lexer //
// **********************************************************************//

class HmlLexer extends Lexer;

options {
  k = 2;
  charVocabulary = '\3'..'\377';
  testLiterals = false;
}

protected ALPHA : ('a'..'z'|'A'..'Z');
protected DIGIT : ('0'..'9') ;

INTLIT         : (DIGIT)+;
STRING          : (ALPHA)+;

COMMENT
  : '/'
    ( ('/') => '/'
      (
        options {greedy = false;}:
        (
          ('\r' '\n') => '\r' '\n' { newline(); }
          | '\r' { newline(); }
          | '\n' { newline(); }
          | ~( '\n' | '\r' )
        )
      )*
      "\\\\"
    | ((~'\n'))* '\n' { newline(); }
    )
  { System.out.println("Ignoring comment: "+getText()); $setType(Token.SKIP); }
  ;

WS
```

```
  : ( ' '
    | '\t'
    | ( "\r\n"  // DOS/Windows
      | '\r'    // Macintosh
      | '\n'    // Unix
      )
      { newline(); }
    )
    { $setType(Token.SKIP); }
  ;

PLUS      : '+'  ;
MINUS     : '-'  ;
MUL       : '*'  ;
DIV       : '/'  ;
MOD       : '%'  ;
POW       : '^'  ;
SEMI      : ';'  ;
LPAREN    : '('  ;
RPAREN    : ')'  ;


// **************************************************************************//
// Tree Parser //
// **************************************************************************//

{import java.lang.Math;}
class HmlTreeWalker extends TreeParser;

expr returns [double r]
  { double a,b; r=0;}
  : #(PLUS  a=expr b=expr)  { r=a+b; }
  | #(MINUS a=expr b=expr)  { r=a-b; }
  | #(MUL   a=expr b=expr)  { r=a*b; }
  | #(DIV   a=expr b=expr)  { r=a/b; }
  | #(MOD   a=expr b=expr)  { r=a%b; }
  | #(POW   a=expr b=expr)  { r=Math.pow(a,b); }
  | #(LPAREN a=expr)        { r=a; }
  | #(SIGN_MINUS a=expr)    { r=-1*a; }
  | #(SIGN_PLUS  a=expr)    { if(a<0)r=0-a; else r=a; }
  | i:INTLIT { r=(double)Integer.parseInt(i.getText()); }
  ;
```