Overview | AsciiMath to LaTeX Converter

1. Project Overview

AsciiMathToLatex takes a math equation written in AsciiMath as input and converts it into valid LaTeX source. The tool first parses the AsciiMath input, outputting syntax errors as they are encountered. It then generates an AST representing the equation which is converted into LaTeX math notation and output as text to the console.

The purpose of the project is to allow for a simpler, more readable syntax for math equations to be used with the superior typesetting abilities of LaTeX.

In terms of formatting, LaTeX is unparalleled for complex math equations. As such, it has become the industry standard and is widely used and supported throughout academia and in math-related fields. Personally, I've been using LaTeX for several years and continue to use it to format math equations and take notes.

However, the syntax to write these math equations is verbose, difficult-to-read, and slow to type. AsciiMath is a language that implements a subset of LaTeX's math functionality and is used for displaying math equations in HTML documents. It uses a much simpler syntax that prefers spaces and parentheses over braces and backslashes and saves the user keystrokes by requiring less explicit instructions.

2. Usage

The AsciiMathToLatex executable takes an equation written in AsciiMath notation as the first and only argument and prints out the converted LaTeX source code. For example, running:

```
./AsciiMathToLatex 'x/y + sqrt(beta)'
```

will output the following LaTeX code:

```
\frac{x}{y} + \sqrt{\beta}
```

Note that you must surround the input equation in single quotes so that spaces are not treated as delimiters in the arguments list.

3. Installation Guide

All the source code for AsciiMathToLatex as well as all the documentation (including this document) were created using Literate, a tool for writing programs using the literate programming method. That means that the source code is contained alongside the documentation within the `.lit` files. Running the Literate program will generate both the `.swift` source files that make up the project and the HTML files that form the documentation. After the `.swift` files are generated by Literate, the Swift compiler is used to create the executable.

Literate
Download and install Literate from their downloads page

Swift Compiler
-OS X users can install the swift compiler, `swiftc`, by installing Xcode
-Linux users can install Swift from the compiler homepage

To build AsciiMathToLatex after installing the above dependencies, simply run `make` from the project directory.

4. Testing

Tests can be run by running the testing script with `swift run_tests.swift`, and their results will be printed out to the console. The tests themselves cover both expected cases and edge cases. For expected cases, the AsciiMathToLatex program is run with valid AsciiMath input, and the output is verified against the known, correct LaTeX source code. For the edge cases, the testing script simply verifies that an error message is produced on bad input. It does not currently verify the quality of the error message, just that the error was caught. The following inputs are tested.

Expected cases:

- 'a/b = c'
- 'a+b <= c^4'
- 'a/b -= alpha_(d in RR)^42 ~= qz sqrt5'
- 'sum_(i=1)^n i^3=((n(n+1))/2)^2'

Edge cases:

- 'a+(c'
- 'b+(c + d/b'

Parser | AsciiMath to LaTeX

1. Description

The parser is initialized with the AsciiMath equation (as text), and encapsulates functionality related to generating a structured representation of the equation and converting that structure to LaTeX. The structure is derived from the grammar definition (in BNR form) presented below.

2. Grammar

The following is a (slightly) modified version of the AsciiMath grammar from the AsciiMath webpage. To the right of the grammar rules are more readable names for each type of symbol, which will be used throughout the program. Note also that for the constant and unary symbols there are far too many to be listed below, the full list can be found on the AsciiMath homepage.

{Grammar definition 2}

```
AsciiMath grammar (BNR)
c ::= [A-Za-z] | greek letters | numbers | other constant symbols     constantSymbols
u ::= sqrt | text | bb | other unary symbols for font commands        unarySymbols
b ::= frac | root | stackrel                                          binarySymbols
l ::= ( | [ | { | (: | {:                                             leftDelimiters
r ::= ) | ] | } | :) | :}                                             rightDelimiters
S ::= c | lEr | uS | bSS                                              simpleExpression
E ::= SE | S/S | S_S | S^S | S_S^S | S | EE                           expression
```

3. Module Structure

The parser class maintains several member variables and exposes two functions in its interface to parse the equation and to convert it to LaTeX.

{parser.swift 3}

```
class Parser {
    {Parser member variables, 4}
    {Parser constructor, 4}
    {Parse input, 5}
    {Convert to latex, 6}
    {Private functions, 7}
}
```

4. Initialization & Member Variables

The class will be initialized with a string representing the AsciiMath equation, and will initialize its own Lexer with that same equation. It was decided to keep the Lexer a private object of the parser to simplify the interface, and to adhere to the principle of "information hiding". Since the caller doesn't need to know about the lexer, it is kept hidden behind the parser's interface.

{Parser constructor 4}

```
init(amEquation: String) {
    self.amEquation = amEquation
    self.lexer = Lexer(amEquation: amEquation)
}
```

Used in section 3

The parser will keep a copy of the equation as well as a Lexer object which will both be initialized as soon as the Parser is. Additionally, the root node of the generated AST will be stored in `astRoot` once parsing is completed. If parsing fails, the `astRoot` will be nil.

{Parser member variables 4}

```
private let amEquation: String!
private var lexer: Lexer!
private var astRoot: Expression?
```

Used in section 3

5. Parsing Input

This function will parse the input, returning false if an error is encountered, and true otherwise. If the parsing is successful, the member variable `astRoot` will store the value of the root node of the AST. Note that the core functionality of the parser is not in this function, but the `getNextExpression` function instead.

{Parse input 5}

```
func parseInput() -> Bool {
    astRoot = getNextExpression()

    if astRoot != nil {
        return true
    } else {
        println("Error parsing input. AST is nil.")
        return false
    }
}
```

Used in section 3

6. Converting to LaTeX

Once the input is validated and parsed, the AST needs to be converted to LaTeX source code. There are several of "cleanup" steps that need to be performed (removing extra spaces, using parentheses that match the equation height etc.). Once the cleanup steps have completed, the resulting LaTeX string is returned.

```
{Convert to latex 6}
```

```
 func convertToLatex() -> String {
     if astRoot != nil {
         var latexStr = astRoot!.toLatexString()
         latexStr = latexStr.stringByReplacingOccurrencesOfString("  ", withString: " ")
         latexStr = latexStr.stringByReplacingOccurrencesOfString("{(", withString: "{")
         latexStr = latexStr.stringByReplacingOccurrencesOfString(")}", withString: "}")
         latexStr = latexStr.stringByReplacingOccurrencesOfString(" }", withString: "}")
         latexStr = latexStr.stringByReplacingOccurrencesOfString(" _", withString: "_")
         latexStr = latexStr.stringByReplacingOccurrencesOfString(" ^", withString: "^")
         latexStr = latexStr.stringByReplacingOccurrencesOfString("(", withString: "\\left(")
         latexStr = latexStr.stringByReplacingOccurrencesOfString(")", withString: "\\right)")
         return latexStr
     } else {
         println("Error converting to LaTeX. AST is nil.")
         return ""
     }
 }
```

Used in section 3

7. Private Functions

The two private functions for getting the next expression or simple expression contain the core functionality of the Parser. These functions use the AsciiMath grammar to generate an AST representing the input equation.

```
{Private functions 7}
```

```
 {Get next simple expression, 8}
 {Get next expression, 9}
```

Used in section 3

8. Simple Expression

This function will return the next simple expression in the text, as per the grammar definition. If no simple expression follows the current location in the string, or if an error is encountered, `nil` is returned.

{Get next simple expression 8}

```
  private func getNextSimpleExpression() -> SimpleExpression? {
      var currentSymbolOpt = lexer.getNextSymbol()

      if currentSymbolOpt == nil {
          println("Error. Expected symbol in simple expression")
          return nil
      }
      if contains(expressionSymbols, currentSymbolOpt!) {
          currentSymbolOpt = lexer.getNextSymbol()
          if (currentSymbolOpt == nil) {
              println("Error. Expected symbol after expression operator")
              return nil
          }
      }
      let currentSymbol = currentSymbolOpt!

      if contains(leftDelimiters, currentSymbol) {
          {Parse lEr rule, 8}
      } else if contains(unarySymbols, currentSymbol) {
          {Parse uS rule, 8}
      } else if contains(binarySymbols, currentSymbol) {
          {Parse bSS rule, 8}
      } else {
          return ConstantSE(str: currentSymbol)
      }
  }
```

Used in section 7

If we are matching to the `lEr` rule (an expression delimited by brackets, braces or parentheses) we need to ensure that after we have the `E`, the correct closing delimiter occurs. If this is indeed the case (as it should be), we want to ignore that closing delimiter by increasing the lexer's current location in the string.

{Parse lEr rule 8}

```
  if let nextExpression = getNextExpression() {
      let delimType = DelimiterType(str: currentSymbol)
      if let nextSymbol = lexer.peekNextSymbol() where nextSymbol == delimType.rightString() {
          lexer.currentLoc += count(delimType.rightString())
          return DelimitedSE(expr: nextExpression, bracketType: delimType)
      } else {
          println("Error. '\(delimType.rightString())' expected after expression '\
  (nextExpression.toString())'")
          return nil
      }
  } else {
      println("Error. expected expression after '\(currentSymbol)'")
      return nil
  }
```

A unary operator followed by a simple expression is a relatively straightforward case. The only possible error is if there is not a simple expression following the operator.

```
{Parse uS rule 8}

  if let nextSimpleExpression = getNextSimpleExpression() {
      return UnarySE(op: currentSymbol, simpleExpr: nextSimpleExpression)
  } else {
      println("Error. Simple expression expected after '\(currentSymbol)'")
      return nil
  }
```

The binary operator is a similar case to the unary operator, except that we require two simple expressions to follow the operator.

```
{Parse bSS rule 8}

  if let nextSimpleExpression = getNextSimpleExpression(),
         finalSimpleExpression = getNextSimpleExpression() {
      return BinarySE(op: currentSymbol, leftSE: nextSimpleExpression, rightSE:
  finalSimpleExpression)
  } else {
      println("Error. Simple expression expected after '\(currentSymbol)'")
      return nil
  }
```

9. Expression

Parsing expressions is a little more complex than simple expressions (as can be expected). All parse rules for expressions start with a simple expression (except `EE`, but that's a special case). So firstly we get the next simple expression and verify it is non-nil. We then check the character following that simple expression. If we're at the end of our string or it's a right delimiter, we just return the simple expression we fetched initially. If it is not one of the special expression symbols ['_','^','/'], then it must be the beginning of a new expression, i.e. the `SE` rule in our grammar. In all these cases, the appropriate Expression object is returned and the function exits.

After addressing all those cases, we know that the next expression must be a fraction or a sub/super-script. When we parse this, we do generate the Expression object representing it, but do not return it immediately. This is because we must check if there is another expression following it. If there is, we return a sequence expression consisting of the first fraction or sub/super-script expression followed by the next expression (which could be any of the possible expression types).

```
{Get next expression 9}

  private func getNextExpression() -> Expression? {
      let simpleExpressionOpt = getNextSimpleExpression()
      if (simpleExpressionOpt == nil) {
          return nil
      }
      let simpleExpression = simpleExpressionOpt!
      var currentExpression: Expression?
      var nextSym = lexer.peekNextSymbol()

      if nextSym == nil || contains(rightDelimiters, String(nextSym!)) {
          return SimpleExpressionE(simpleExpr: simpleExpression)
      }

      if !contains(expressionSymbols, String(nextSym!)) {
          {Parse SE rule, 9}
      }

      if let nextSimpleExpression = getNextSimpleExpression() {
          switch (nextSym!) {
          case "/":
              currentExpression = FractionE(top: simpleExpression, bottom: nextSimpleExpression)

          case "_":
              if (lexer.peekNextSymbol() == "^") {
                  {Parse S_S^S rule, 9}
              } else {
                  currentExpression = SubscriptE(base: simpleExpression, sub:
  nextSimpleExpression)
              }

          case "^":
              currentExpression = SuperscriptE(base: simpleExpression, superscript:
  nextSimpleExpression)

          default: break
          }

          {Check for EE rule, 9}

      }
      return currentExpression
  }
```

Used in section 7

The most common type of expression consists of a simple expression followed by another expression. In this case, the only possible error (since we already verified the simple expression) would be no expression following it.

```
{Parse SE rule 9}

  if let nextExpression = getNextExpression() {
      return SimpleSequenceE(simpleExpr: simpleExpression, expr: nextExpression)
  } else {
      println("Error. Expression in SE case expected following '\(simpleExpression.toString())'")
      return nil
  }
```

This is a more complex case of sub or superscript. Note that the subscript must come first, as per AsciiMath standard. The possible error (since we've already verified the first two simple expressions) is that the third simple expression, the superscript, does not exist.

```
{Parse S_S^S rule 9}

  if let finalSimpleExpression = getNextSimpleExpression() {
      currentExpression = SubSuperscriptE(base: simpleExpression, sub: nextSimpleExpression,
  superscript: finalSimpleExpression)
  } else {
      println("Error. Simple expression expected following '\(simpleExpression.toString())_\
  (nextSimpleExpression.toString())^'")
      return nil
  }
```

If we haven't reached the end of our string, there is a possibility that there is another expression following the one we just parsed. If this is the case, we return the appropriate sequence expression. We must also check that the next character is not a right delimiter, as right delimiters indicate the end of an expression contained within the `lEr` simple expression rule.

```
{Check for EE rule 9}

  if currentExpression != nil && lexer.currentLoc < count(amEquation) - 1
  && !contains(rightDelimiters, lexer.peekNextSymbol()!) {
      if let nextE = getNextExpression() {
          return SequenceE(e1: currentExpression!, e2: nextE)
      } else {
          println("Error. No expression found after '\(currentExpression)', but haven't reached
  end of equation.")
          return currentExpression
      }
  }
```

Utilities | AsciiMath to LaTeX

1. Introduction

A common task used in the program is to look at specific characters within a string, and ranges of the string. To simplify this, an extension to the String class adds subscript functionality to improve the syntax of performing these operations.

2. Module Structure

The utility functions are applied as extensions to the built-in String class, so they may be used on standard Strings.

{util.swift 2}

```
extension String {
    {Subscript strings, 3}
}
```

3. Subscript Strings

The following to methods provide a simplified syntax for accessing a character at a given index in a string, and to access a substring of a string.

{Subscript strings 3}

```
subscript(i: Int) -> Character {
    return Array(self)[i]
}
subscript(r: Range<Int>) -> String {
    return String(Array(self)[r.startIndex ..< r.endIndex])
}
```

Used in section 2

Lexer | AsciiMath to LaTeX

1. Description

The lexer module accesses the string at a lower level than the parser, on a character-by-character basis. There are to public functions used by the Parser. The simpler of the two allows the caller to see what the next character in the string is (relative to the current location) without updating the current location. The main function used by the Parser returns the next symbol (potentially multiple characters) in the string.

It is the lexer that stores all the possible symbols in AsciiMath, as per their homepage. These are public, as they are used by the parser.

2. Module Structure

The majority of the module is encapsulated within the Lexer class, with the exception of the symbol declaration. Since the symbols are accessed by the Parser and the expression types, these remain public.

{lexer.swift 2}

```
class Lexer {
    {Lexer member variables, 3}
    {Lexer constructor, 4}
    {Get symbol, 5}
    {Get multiple character symbol, 5}
    {Peek character, 6}
    {Peek symbol, 7}
    {Get character, 8}
}
{Symbols, 9}
```

3. Member Variables

The only public member variable of the Lexer represents the current location in the string (as an index). As the lexer returns symbols in the string, the current location increases.

{Lexer member variables 3}

```
private let amEquation: String!
private let allSymbols: [String]!
var currentLoc: Int = 0
```

Used in section 2

4. Constructor

The constructor simply sets the AsciiMath equation string member variable, and takes all the different types of symbols and combines them into one array to simplify checking for a match.

{Lexer constructor 4}

```
  init(amEquation: String) {
      self.amEquation = amEquation
      allSymbols = [unarySymbols, binarySymbols, leftDelimiters, rightDelimiters,
  expressionSymbols,
                        greekSymbols, relationSymbols, operationSymbols, miscSymbols,
  arrowSymbols].flatMap { $0 }
  }
```

Used in section 2

5. Get Symbol

There are two main cases for a symbol - it is either a multi-character symbol (such as `alpha` or `frac`), or it is a single character (such as `n` or `/`). The function first retrieves the next character, and then checks if there are any multi-character symbols that match the substring starting at the current index. If there are, the longest one is returned. If not, then the current character is returned. Note that if the current character is a digit, we check if there are more digits following it and return the multiple digit number rather than just the first digit if that's the case.

{Get symbol 5}

```
  func getNextSymbol() -> String? {
      let currentCharacter = getNextCharacter()
      if (currentCharacter == nil) {
          return nil
      }

      var currentSymbol = String(currentCharacter!)
      if let multicharSymbol = getMulticharSymbol() {
          return multicharSymbol
      } else {
          {Check for multiple digit number, 5}
          return currentSymbol
      }
  }
```

Used in section 2

If we don't check for multi-digit numbers and return the full number, then multiple digit numbers will be treated as separate simple expressions, and will not be grouped properly. For example, `a^12` would be interpreted as `a^{1} 2` rather than the correct `a^{12}`.

```
{Check for multiple digit number 5}

  if currentSymbol.toInt() != nil {
      var nextChar = peekNextCharacter()
      while (nextChar != nil && String(nextChar!).toInt() != nil) {
          currentSymbol += String(nextChar!)
          currentLoc++
          nextChar = peekNextCharacter()
      }
  }
```

If a multiple character symbol is matched at the current location, it is returned, otherwise `nil` is returned. Note that it is the longest matching symbol that is returned, rather than the first one. Since some symbols are prefixes to longer symbols, we need to ensure we take the longest. The performance of this function could be significantly improved, but it was left in its current form since the AsciiMath equations passed to the program are usually very short, so the program executes very quickly nonetheless. It is planned to improve the performance of this function in the future.

```
{Get multiple character symbol 5}

  private func getMulticharSymbol() -> String? {
      var longestMatch = ""
      for symbol in allSymbols {
          let symLength = count(symbol)

          if currentLoc-1 + symLength <= count(amEquation)
          && amEquation[currentLoc-1..<currentLoc+symLength-1] == symbol
          && symLength > count(longestMatch) {
              longestMatch = amEquation[currentLoc-1..<currentLoc+symLength-1]
          }
      }

      if longestMatch != "" {
          currentLoc += count(longestMatch) - 1
          return longestMatch
      }
      return nil
  }
```

Used in section 2

6. Peek Character

The peek function allows the caller to see what the next character is (or if it is `nil`) without modifying the current location in the string.

{Peek character 6}

```
  func peekNextCharacter() -> Character? {
      if (currentLoc >= count(amEquation)) {
          return nil
      }
      return amEquation[currentLoc]
  }
```

Used in section 2

7. Peek Symbol

The peek function allows the caller to see what the next symbol is (or if it is `nil`) without modifying the current location in the string.

{Peek symbol 7}

```
  func peekNextSymbol() -> String? {
      if (currentLoc >= count(amEquation)) {
          return nil
      }
      let currentLocPre = currentLoc
      let nextSymbol = getNextSymbol()
      currentLoc = currentLocPre
      return nextSymbol
  }
```

Used in section 2

8. Get Character

Get character simply returns the next character in the string, or `nil` if the current location passes at the end of the string.

{Get character 8}

```
  private func getNextCharacter() -> Character? {
      if (currentLoc >= count(amEquation)) {
          return nil
      } else {
          let c = amEquation[currentLoc];
          currentLoc++;
          return c
      }
  }
```

Used in section 2

9. Symbols

The symbols supported by AsciiMath are outlined on the homepage. They are declared as public as they are needed by the constant expression type and the parser.

{Symbols 9}

```
let unarySymbols = ["sqrt","text","bb","hat","bar","ul","vec","dot","ddot"]
let binarySymbols = ["frac","root","stackrel"]
let leftDelimiters = ["(","[","{"]
let rightDelimiters = [")","]","}"]
let expressionSymbols = ["^","_","/"]

// constants
let greekSymbols = ["alpha","beta","chi","delta","Delta","epsilon","varepsilon",
                    "eta","gamma","Gamma","iota","kappa","lambda","Lambda","mu",
                    "nu","omega","Omega","phi","Phi","varphi","pi","Pi","psi",
                    "Psi","rho","sigma","Sigma","tau","theta","Theta","vartheta",
                    "upsilon","xi","Xi","zeta"]
let relationSymbols = ["!=","<=",">=","-<",">-","in","!in","sub","sup","sube","supe",
                       "-=","~=","~~","prop"]
let operationSymbols = ["**","***","//","\\\\","xx","-:","@","o+","ox","o.","sum",
                        "prod","^^","^^^","vv","vvv","nn","nnn","uu","uuu"]
let miscSymbols = ["int","oint","del","grad","+-","O/","oo","aleph","/_",":.","|...|",
                   "|cdots|","vdots","ddots","|\\|","|quad|","diamond","square","|__",
                   "__|","|~","~|","CC","NN","QQ","RR","ZZ"]
let logicalSymbols = ["and","or","not","=>","if","iff","AA","EE","_|_","TT","|--","|=="]
let arrowSymbols = ["uarr","darr","rarr","->","|->","larr","harr","rArr","lArr","hArr"]
```

Used in section 2

Types | AsciiMath to LaTeX

1. Description

Each of the possible expression and simple expression "types" corresponding to an option in the grammar definition is represented by a `struct` that stores the appropriate sub-expressions and provides funtions to output a string representation for debugging, as well as a LaTeX representation for final output.

As well as the above types, there is also a type representing a delimiter. This is also modelled as a `struct`, and provides functions to output the left or right delimter of a given type. This is used to simplify the verification of matching delimiters in parsing.

The constant symbols in AsciiMath are often much shorter than their LaTeX counterparts, and as such a facility to retrieve the correct LaTeX symbol for a given constant is required. This is implemented as a switch with cases for each symbol that differs from its LaTeX counterpart. For symbols that are the same, such as `alpha`, a default case prepends a backslash and adds a space at the end.

2. Module Structure

Since the types declared in this file are used throughout the project, they are not embedded within a class. The function to retrieve the LaTeX represenation of a constant symbol is embedded within the constant simple expression type, as it is only used within that type.

```
{expressiontypes.swift 2}

  {Expression type interface, 3}
  {Simple expressions, 3}
  {Expressions, 4}
  {Bracket type, 5}
```

3. Interface

Although both simple expressions and expressions have identical interfaces, they are presented as separate protocols to improve the readability of the program. This adds some code duplication, but it allows the reader to better reason about the type of a given object. The interface that they both share simply provides functions to retrieve the string representation of the expression (for debugging) as well as the LaTeX equivalent.

```
{Expression type interface 3}

  protocol SimpleExpression {
      func toString() -> String
      func toLatexString() -> String
  }
  protocol Expression {
      func toString() -> String
      func toLatexString() -> String
  }
```

Used in section 2

The following types for simple expressions match the four possibilities for a simple expression in the AsciiMath gramamr.

```
{Simple expressions 3}

  {Constant SE, 3}
  {Delimited SE, 3}
  {Unary SE, 3}
  {Binary SE, 3}
```

Used in section 2

Constant simple expressions (`c` in the grammar) include numbers (potentially multi-digit), greek symbols, and single characters. Constants form the "base" of an equation. This type contains a function to convert the constant symbol from AsciiMath to LaTeX. For example, the AsciiMath symbol "oo" is used to represent infinity, which in LaTeX is "\infty".

```
{Constant SE 3}

  struct ConstantSE: SimpleExpression {
      var str = ""
      init(str: String) {
          self.str = str
      }
      func toString() -> String {
          return str
      }
      func toLatexString() -> String {
          return latexForConstantSymbol(str)
      }
      {Constant symbols to Latex, 6}
  }
```

Delimited simple expressions consist of an expression surrounded by matching delimiters (parentheses, braces etc.). Note that the DelimiterType used by the simple expression is not contained within the type as is the constant symbol conversion function discussed above. This is because the DelimiterType is used by other modules, so must remain public.

```
{Delimited SE 3}

 struct DelimitedSE: SimpleExpression {
     var expression: Expression!
     var bracketType: DelimiterType!

     init(expr: Expression, bracketType: DelimiterType) {
         self.expression = expr
         self.bracketType = bracketType
     }
     func toString() -> String {
         return "\(bracketType.leftString())\(expression.toString())\(bracketType.rightString())"
     }
     func toLatexString() -> String {
         var leftDelim = bracketType.leftString()
         var rightDelim = bracketType.rightString()
         if bracketType == .Brace {
             leftDelim = "\\{"
             rightDelim = "\\}"
         }
         return "\(leftDelim)\(expression.toLatexString())\(rightDelim)"
     }
 }
```

A unary simple expression is simply a unary operator (e.g. `sqrt`) followed by another simple expression.

```
{Unary SE 3}

 struct UnarySE: SimpleExpression {
     var simpleExpression: SimpleExpression
     var unaryOperator = ""

     init(op: String, simpleExpr: SimpleExpression) {
         self.unaryOperator = op
         self.simpleExpression = simpleExpr
     }
     func toString() -> String {
         return "\(unaryOperator)\(simpleExpression.toString())"
     }
     func toLatexString() -> String {
         return "\\\(unaryOperator){\(simpleExpression.toLatexString())}"
     }
 }
```

A binary simple expression is simply a binary operator (e.g. `frac`) followed by two simple expressions.

```
{Binary SE 3}

  struct BinarySE: SimpleExpression {
      var simpleExpressionLeft: SimpleExpression
      var simpleExpressionRight: SimpleExpression
      var binaryOperator = ""

      init(op: String, leftSE: SimpleExpression, rightSE: SimpleExpression) {
          self.binaryOperator = op
          self.simpleExpressionLeft = leftSE
          self.simpleExpressionRight = rightSE
      }
      func toString() -> String {
          return "\(binaryOperator)\(simpleExpressionLeft.toString())\
  (simpleExpressionRight.toString())"
      }
      func toLatexString() -> String {
          return "\\\(binaryOperator){\(simpleExpressionLeft.toLatexString())}{\
  (simpleExpressionRight.toLatexString())}"
      }
  }
```

4. Expressions

There are more types of expressions than simple expressions, as expressions form the base of the grammar (i.e. the root node of the AST is an expression).

```
{Expressions 4}

  {Simple expression E, 4}
  {Simple sequence E, 4}
  {Sequence E, 4}
  {Fraction E, 4}
  {Subscript E, 4}
  {Superscript E, 4}
  {SubSuperscript E, 4}
```

Used in section 2

The simplest type of expression consists only of a simple expression.

{Simple expression E 4}

```
 struct SimpleExpressionE: Expression {
     var simpleExpression: SimpleExpression
     init(simpleExpr: SimpleExpression) {
         self.simpleExpression = simpleExpr
     }
     func toString() -> String {
         return simpleExpression.toString()
     }
     func toLatexString() -> String {
         return simpleExpression.toLatexString()
     }
 }
```

The simple sequence consists of a simple expression followed by an expression. This recursive definition provides the backbone for most equations, as a long series of constant symbols is represented by it.

{Simple sequence E 4}

```
 struct SimpleSequenceE: Expression {
     var simpleExpression: SimpleExpression
     var expression: Expression

     init(simpleExpr: SimpleExpression, expr: Expression) {
         self.simpleExpression = simpleExpr
         self.expression = expr
     }
     func toString() -> String {
         return "\(simpleExpression.toString())\(expression.toString())"
     }
     func toLatexString() -> String {
         return "\(simpleExpression.toLatexString())\(expression.toLatexString())"
     }
 }
```

The sequence expression represents an expression followed by another expression. Without this rule, equations with a "traditional" expression (i.e. fraction, sub/super-script) would not be parsed after the expression.

```
{Sequence E 4}

 struct SequenceE: Expression {
     var e1: Expression
     var e2: Expression
     init(e1: Expression, e2: Expression) {
         self.e1 = e1
         self.e2 = e2
     }
     func toString() -> String {
         return "\(e1.toString())\(e2.toString())"
     }
     func toLatexString() -> String {
         return "\(e1.toLatexString())\(e2.toLatexString())"
     }
 }
```

An expression that greatly simplifies the LaTeX syntax for fractions, the fraction expression consists of a simple expression followed by a forward slash and another simple expression. Note that to have complex expressions in the fraction they must be delimited simple expressions.

```
{Fraction E 4}

 struct FractionE: Expression {
     var simpleExpressionTop: SimpleExpression
     var simpleExpressionBottom: SimpleExpression

     init(top: SimpleExpression, bottom: SimpleExpression) {
         self.simpleExpressionTop = top
         self.simpleExpressionBottom = bottom
     }
     func toString() -> String {
         return "\(simpleExpressionTop.toString())/\(simpleExpressionBottom.toString())"
     }
     func toLatexString() -> String {
         return "\\frac{\(simpleExpressionTop.toLatexString())}{\
 (simpleExpressionBottom.toLatexString())}"
     }
 }
```

The subscript expression is simply a simple expression followed by an underscore, and another simple expression. Note that complex expressions must be delimited simple expressions.

```
{Subscript E 4}
```

```
struct SubscriptE: Expression {
    var simpleExpressionBase: SimpleExpression
    var simpleExpressionSubscript: SimpleExpression

    init(base: SimpleExpression, sub: SimpleExpression) {
        self.simpleExpressionBase = base
        self.simpleExpressionSubscript = sub
    }
    func toString() -> String {
        return "\(simpleExpressionBase.toString())_\(simpleExpressionSubscript.toString())"
    }
    func toLatexString() -> String {
        return "\(simpleExpressionBase.toLatexString())_{\
(simpleExpressionSubscript.toLatexString())}"
    }
}
```

The superscript expression is simply a simple expression followed by a caret, and another simple expression. Note that complex expressions must be delimited simple expressions.

```
{Superscript E 4}
```

```
struct SuperscriptE: Expression {
    var simpleExpressionBase: SimpleExpression
    var simpleExpressionSuperscript: SimpleExpression

    init(base: SimpleExpression, superscript: SimpleExpression) {
        self.simpleExpressionBase = base
        self.simpleExpressionSuperscript = superscript
    }
    func toString() -> String {
        return "\(simpleExpressionBase.toString())^\(simpleExpressionSuperscript.toString())"
    }
    func toLatexString() -> String {
        return "\(simpleExpressionBase.toLatexString())^{\
(simpleExpressionSuperscript.toLatexString())}"
    }
}
```

This expression combines a subscript and a superscript, with the subscript being declared first. Note that complex expressions must be delimited simple expressions.

```
{SubSuperscript E 4}

 struct SubSuperscriptE: Expression {
     var simpleExpressionBase: SimpleExpression
     var simpleExpressionSubscript: SimpleExpression
     var simpleExpressionSuperscript: SimpleExpression

     init(base: SimpleExpression, sub: SimpleExpression, superscript: SimpleExpression) {
         self.simpleExpressionBase = base
         self.simpleExpressionSubscript = sub
         self.simpleExpressionSuperscript = superscript
     }
     func toString() -> String {
         return "\(simpleExpressionBase.toString())_\(simpleExpressionSubscript.toString())^\
 (simpleExpressionSuperscript.toString())"
     }
     func toLatexString() -> String {
         return "\(simpleExpressionBase.toLatexString())_{\
 (simpleExpressionSubscript.toLatexString())}^{\(simpleExpressionSuperscript.toLatexString())}"
     }
 }
```

5. Delimiter Type

Representing delimiters by a type (rather than a string) allows the programmer to easily access the corresponding delimiter of a given delimiter. If the programmer mistakenly tries to initialize a delimiter type with an unspported symbol, an error message is output and the type defaults to parentheses.

```
{Bracket type 5}

  enum DelimiterType {
      case Paren, Bracket, Brace
      func leftString() -> String {
          switch self {
          case Paren: return "("
          case Bracket: return "["
          case Brace: return "{"
          }
      }
      func rightString() -> String {
          switch self {
          case Paren: return ")"
          case Bracket: return "]"
          case Brace: return "}"
          }
      }
      init(str: String) {
          switch(str) {
          case "(",")": self = Paren
          case "[","]": self = Bracket
          case "{","}": self = Brace
          default:
              println("Error. Trying to initialize DelimiterType with unsupported bracket '\
  (str)'")
              self = Paren
          }
      }
  }
```

Used in section 2

6. Constant Symbols to LaTeX

The constant symbols in AsciiMath use a simpler syntax than their LaTeX counterparts. This function returns the corresponding LaTeX representation for the input symbol. In the case where the LaTeX symbol is the same (with a backslash at the beginning), no explicit case in the switch statement is required.

```
{Constant symbols to Latex 6}

  private func latexForConstantSymbol(symbol: String) -> String {
      var latexSymbol = ""
      switch(symbol) {
      // operation symbols
      case "*": latexSymbol = "cdot"
      case "**": return "*"
      case "***": latexSymbol = "star"
      case "//": return "/"
      case "\\\\": return "\\ "
      case "xx": latexSymbol = "times"
      case "-:": latexSymbol = "div"
      case "@": latexSymbol = "circ"
      case "o+": latexSymbol = "oplus"
      case "ox": latexSymbol = "otimes"
      case "o.": latexSymbol = "odot"
      case "^^": latexSymbol = "land"
      case "^^^": latexSymbol = "bigwedge"
      case "vv": latexSymbol = "lor"
      case "vvv": latexSymbol = "bigvee"
      case "nn": latexSymbol = "intersection"
      case "nnn": latexSymbol = "bigcap"
      case "uu": latexSymbol = "union"
      case "uuu": latexSymbol = "bigcup"

      // miscellaneous symbols
      case "del": latexSymbol = "delta"
      case "grad": latexSymbol = "nabla"
      case "+-": latexSymbol = "pm"
      case "O/": latexSymbol = "Theta"
      case "oo": latexSymbol = "infty"
      case "/_": latexSymbol = "angle"
      case ":.": latexSymbol = "therefore"
      case "|cdots|": return "|\\cdots"
      case "|quad|": return "|\\quad|"
      case "diamond": latexSymbol = "Diamond"
      case "square": latexSymbol = ""
      case "|__": latexSymbol = "lfloor"
      case "__|": latexSymbol = "rfloor"
      case "|~": latexSymbol = "lceil"
      case "~|": latexSymbol = "rceil"
      case "CC": latexSymbol = "mathbb{C}"
      case "NN": latexSymbol = "mathbb{N}"
      case "QQ": latexSymbol = "mathbb{Q}"
      case "RR": latexSymbol = "mathbb{R}"
      case "ZZ": latexSymbol = "mathbb{Z}"

      // relation symbols
      case "!=": latexSymbol = "neq"
      case "<=": latexSymbol = "leq"
      case ">=": latexSymbol = "geq"
      case "-<": latexSymbol = "prec"
      case ">-": latexSymbol = "succ"
      case "!in": latexSymbol = "notin"
      case "sub": latexSymbol = "subset"
      case "sube": latexSymbol = "subseteq"
      case "sup": latexSymbol = "supset"
      case "supe": latexSymbol = "supseteq"
      case "-=": latexSymbol = "equiv"
      case "~=": latexSymbol = "cong"
      case "~~": latexSymbol = "approx"
      case "prop": latexSymbol = "propto"

      // logical symbols
      case "and": latexSymbol = ""
      case "or": latexSymbol = ""
      case "not": latexSymbol = ""
      case "=>": latexSymbol = ""
      case "if": latexSymbol = ""
      case "iff": latexSymbol = ""
```

```
    case "AA": latexSymbol = ""
    case "_|_": latexSymbol = ""
    case "TT": latexSymbol = ""
    case "|--": latexSymbol = ""
    case "|==": latexSymbol = ""

    // arrow symbols
    case "uarr": latexSymbol = "uparrow"
    case "darr": latexSymbol = "downarrow"
    case "rarr": latexSymbol = "rightarrow"
    case "->": latexSymbol = "rightarrow"
    case "|->": latexSymbol = "longmapsto"
    case "larr": latexSymbol = "leftarrow"
    case "harr": latexSymbol = "leftrightarrow"
    case "rArr": latexSymbol = "Rightarrow"
    case "lArr": latexSymbol = "Leftarrow"
    case "hArr": latexSymbol = "Leftrightarrow"

    default: latexSymbol = symbol
    }
    // single letters shouldn't have \ prepended
    if count(latexSymbol) == 1 || latexSymbol.toInt() != nil {
        return symbol
    }
    return "\\\(latexSymbol) "
}
```

Used in section 3