Abstract Syntax Trees
for Khan Exercises

January 4, 2012

```
Authors: Jeff Dyer <jeffdyer@acm.org>
         Mathias Hiron <mathias@khanacademy.org>
```

```
History: 04-Jan-2012, second draft, reviewed by Mathias
         03-Jan-2012, first draft, unreviewed
```

1 Overview

In this note we define an abstract syntax tree (AST) form for encoding information used in Khan exercises. We also define a mapping from ASTs to LaTeX formatted strings. ASTs and LaTeX have wide applicability in the Khan exercise framework, and so by defining and implementing a core library for handling them we avoid duplication of effort and gain the benefits normally associated by software reuse.

2 Abstract Syntax Trees

We use trees to represent expressions, equations, and other structured data that underlies Khan exercises. The trees consist of nodes containing an operator string and zero or more child nodes.

The nodes are encoded as JavaScript objects of the form:

{**op**: *opstr*, **args**: [$nd1, \ldots ndN$]}

where *opstr* is an *operator string* and $nd1, \ldots ndN$ is a sequence of child nodes.

An *operator string* is a string value that, along with the number of arguments, distinguishes one node kind from another. AST nodes get their meaning from the parsing, formatting and evaluating procedures that construct and interpret them, and otherwise have no special meaning within the AST definition and implementation.

Primitive (non-object) JavaScript values are used to represent literal numbers and strings in the AST. Variables are encoded as nodes with the unary "var" operator with a literal string operand.

We define a core set of operators here, but this set is not fixed. As new exercises require new operators, additional strings to be contrived to represent those operators, and interpreters can be defined to format and evaluate nodes with those operators.

3 Predefined Operators

The following set of operators are implemented by the AST module (see below). Additional operators may be added to this core set over time, and others may be added by problem specific utility modules.

Operator	**Sample**	**Comment**
`+`	$x + y$	addition, unary plus
`-`	$x - y$	subtraction, unary minus
`times`	$x \times y$	multiplication
`div`	$x \div y$	division
`frac`	$\frac{x}{y}$	fraction
`^`	x^y	exponent
`sqrt`	$\sqrt{x}$	unary square root
`sqrt`	$\sqrt[n]{x}$	binary square root
`pm`	$x \pm y$	plus or minus, unary plus or minus
`sin`	$\sin\theta$	sine
`cos`	$\cos\theta$	cosine
`tan`	$\tan\theta$	tangent
`sec`	$\sec\theta$	secant
`ln`	$\ln x$	natural logarithm
`()`	$(expr)$	parenthesis
`var`	x	variable (unknown)

4 LaTeX Translation

In addition to the internal AST form, we define a mapping from ASTs to LaTeX. The following table shows the mapping between the AST nodes and LaTeX.

AST	**LaTeX**
`{op: "+", args: [1, 2]}`	`1+2`
`{op: "-", args: [1, 2]}`	`1-2`
`{op: "times", args: [1, 2]}`	`1 \times 2`
`{op: "div", args: [1, 2]}`	`1 \div 2`
`{op: "frac", args: [1, 2]}`	`\frac{1}{2}`
`{op: "^", args: [1, 2]}`	`1^2`
`{op: "sqrt", args: [9]}`	`\sqrt{9}`
`{op: "sqrt", args: [3, 8]}`	`\sqrt[3]{8}`
`{op: "pm", args: [2, 1]}`	`2 \pm 1`
`{op: "sin", args: [0]}`	`\sin{0}`
`{op: "cos", args: [0]}`	`\cos{0}`
`{op: "tan", args: [0]}`	`\tan{0}`
`{op: "sec", args: [0]}`	`\sec{0}`
`{op: "ln", args: [1]}`	`\ln{1}`
`{op: "()", args: [10]}`	`(10)`
`{op: "var", args: ["x"]}`	`\var{x}`

The AST library provides functions for translating between ASTs and LaTeX formatted strings. These formatting and parsing algorithms conform to common practice including:

- Insert `\left` and `\right` to auto size brackets.
- Rewrite addition of a negative number as subtraction.
- Elide `\times` when it occurs before a non-numeric symbol during formatting, and construct a `\times` node when a non-numeric symbol occurs after a non-operator symbol.
- Insert parenthesis when formatting a binary expression of lower precedence that occurs in an expression of higher precedence.
- (more here)

5 AST API

The AST API is implemented as a module that can be loaded into an exercise. All functions are instance methods of the **ast** property added to **KhanUtil**, and therefore are called through that property (e.g. **KhanUtil.ast.fromLaTeX**("1+2"), or **ast.fromLaTeX**("1+2") when called from exercise HTML that loads the AST module).

Below is the list of API methods implemented by `ast.js`.

Name	**Description**
fromLaTeX(*str*)	Return an AST node constructed from a LaTeX formatted string
toLaTeX(*node*)	Return the LaTeX formatted string correponding to an AST node
eval(*node*)	Return the result of evaluating an AST
intern(*node*)	Add an AST node and its child nodes to the node pool and return the root node id (nid)
node(*nid*)	Reconstitute an AST from its node id