Abstract Syntax Trees for Khan Exercises

January 3, 2012

Author(s): Jeff Dyer <jeffdyer@acm.org>
Status: Unreviewed
History: 03-Jan-2012, first draft

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 the internal AST form to several surface syntaxes useful for formatting and constructing ASTs. These data structures have wide applicability in the Khan exercise framework, and so by defining and implementing a core library we avoid duplication of effort and gain the benefits normally associated by software reuse.

We separate the management of ASTs from their construction and interpretation. This allows an AST kind to be interpreted differently for different uses. Furthermore, it allows us to easily add new node kinds without having to modify the core AST code.

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 this form:

{op: *opstr*, **args**: [*nd*1,...*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.

We define a core set of operators but this set is not fixed. As new exercises require new opertors, 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 Concrete Syntax

In addition to the internal AST form, we define three concrete syntaxes useful for specifying and displaying AST based content. The three forms are LaTeX, plain text and, for convenience of construction, square bracketed S-expressions. The following table shows the mapping between the core AST nodes and these three formats.

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

The AST library provides functions for translating between the three alternative formats and the internal AST format. Extensions can override these translations as necessary. 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

We provide an API for constructing, formatting, evaluating, and interning ASTs. Some of these functions take a optional *model* object which interprets the meaning of operators that are not defined by the AST library, or which override those that are. When the *model* parameter is given, the AST function delegates the call to that object, which may call back to the AST function to handle operators it does not distinguish.

The AST object 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. **ast.fromText**("1+2"), or simply **fromText**("1+2") when called from exercise HTML that loads the AST module).

Name	Description
fromSExp(*sexp, model*)	Return an AST node constructed from an s-expression formatted array
fromText(*str, model*)	Return an AST node constructed from a plain text formatted string
fromLaTex(*str, model*)	Return an AST node constructed from a LaTeX formatted string
toSExp(*node, model*)	Return the s-expression corresponding to an AST node
toLaTex(*node, model*)	Return the LaTeX formatted string correponding to an AST node
toText(*node, model*)	Return the plain text formatted string corresponding to an AST node
eval(*node, model*)	Return the result of evaluating an AST given a model
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