code directly.

The effort to implement this was roughly 1 day from start to finish. Consider the effort it likely took the Mathematica folks to get to this level of capability.

Modeling Algebra as a text-based language with DMS

Algebra as printed in textbooks has fancy fonts, super- and sub-scripts, Greek characters, and if you get to calculus, rather unique symbols such as the integral sign in its many variants. Algebra as written by mathematicians on paper skips the fonts but Greek and integral signs are still standand. Since we want to do algebra with the computer, we choose to make our example algebraic notation live within the constraints of standard computer text.

We outlaw fonts by using unadorned plain text. Numbers need to look like strings of digits; that's easy. Variable names on paper are often written as single characters (and that's why Greek is popular: it increases the set of possible names one can use). In a computer world, we choose to model variable names much like identifiers in programming languages, consisting of a leading alphabetic character followed by alphabetic characters or digits. Following the math world lead, we use the Unicode character set, and take the definition of "alphabetic" from the Unicode standard; this lets us use a-z, Greek, or Chinese variable names. Superscripts (well, exponents) we model by the usual computer-language artifice of using the ^ (caret) character, but we allow the Unicode up-arrow symbol ↑ (\u2191) to be nice. Function calls are written using the usual computer notation: *name***(***expression***)**. Like most computer languages, we make multiplication explicit using the operator *****. (It would be straightforward to define a grammar with the multiplication operator implicit as it is in most mathematical equations.) A typical equation written using our text equivalent of algebra is

```
x^e+2*x^2
```

The d/dx operator for calculus we model with special syntax:
D*name***:***expression*
with the variable explicitly specified. Following that lead, we model (indefinite) integrals with special syntax:
I*expression***D***name*
(allowing the Unicode integral sign character ∫ (\u222b) as a alternative to **I**). Definite integrals have additional special syntax to provide placeholder for low and upper bound expressions. A typical derivative equation would be written as:

```
Dz:(z^2+x*log(z))
```

Finally, we added a special notation to indicate that we want to substitute a formula for a particular variable in an expression:

S*name***:***expression*

We don't use the integral or substitution formulas in examples on this page, but they are there so we can play with this domain more in the future. With DMS, additional DSL syntax to handle the future is cheap.

You should easily see all these decisions in the formal description of the Algebra domain provided to DMS, below.

DMS Domain Definition Elements Necessary

The following formal descriptions are needed for DMS to parse/transform/prettyprint algebra:

- A lexical definition: this defines the elements ("tokens") of our algebra language
- A grammar definition: this defines the shape of algebra equation in terms of allowable sequences of tokens
- A prettyprinter definition: this defines how to print an instance of algebra equations out as text
- Transformation Rules: these define how algebra equations can be symbolically transformed using source-to-source transformations.

Lexical Definition for Algebra domain

What follows is the exact definition used for the Algebra domain. It defines *tokens* (or lexemes) which make up the elements of the input language for our Algebra domain. Tokens are defined by a token name, a regular expression over Unicode that determines the allowable sequence of characters for the token, an internal representation for the token value (if any), and a conversion procedure to generate that internal representation (if any).

Token defintions are provided in the format:

#token *token_name* **[***internal_representation***]** **"***regular_expression***"**
<< *conversion_code* >>

Literal tokens, those that have essentially only one spelling, are by convention named **'**<*token_spelling*>**'**, e.g., '+', 'S' and even '('; this gives the prettyprinter the default needed to print the token at a later time. Tokens that have variable spelling and interesting content and by convention get SHOUTED names (e.g, NUMBER and NAME for Algebra). Note how we defined NAME using a DMS library-provided definition to match our desired text model. Such tokens need code to convert the characters comprising the instance token into machine-native data structures such as binary numbers, float values, and/or actual

character strings. The conversion code must also capture lexical formatting data (leading zero count on decimal numbers, number radix) that will be needed to regenerate text for that lexeme by the prettyprinter. The conversion code is written in DMS's PARLANSE programming language. Note that a library of predefined conversions is defined by DMS, that covers most interesting cases.

```
-- Algebra.lex
-- Copyright (C) 2010 Semantic Designs; All Rights Reserved

%%Algebra

#include "DMSLexical/Library/Unicode.lex"

#macro newline "\u000D \u000a | \u000d | \ u000a"

#skip "([\s]|<newline>)+"

#token ':' "\:"
#token '=' "\="
#token ',' "\,"

#token '+' "\+"
#token '-' "\-"
#token '*' "\*"
#token '/' "\/"
#token '^' "[\^ \u2191]"
#token '(' "\("
#token ')' "\)"
#token NUMBER [NATURAL] " [0-9]+ "
  << (local [format LiteralFormat:NaturalLiteralFormat]
       (;; (= ?:Lexeme:Literal:Natural:Value
              (ConvertDecimalTokenStringToNatural (. format) ? 0 0))
           (= ?:Lexeme:Literal:Natural:Format
              (LiteralFormat:MakeCompactNaturalLiteralFormat format))
       );;
     )local
  >>

#token 'S' "S" -- "substitute" operator
#token 'D' "D|\u2202" -- "derivative" operator
#token 'I' "I|\u222b" -- "integrate" operator

#token NAME [STRING] "<UnicodeIsLetterLowercase>(<UnicodeIsLetterLowercase>#[0-9])*"
  <<  (;; (ConvertTokenStringToString (. ?:Lexeme:Literal:String:Value) ? 0 0)
           (= ?:Lexeme:Literal:String:Format
              (LiteralFormat:MakeCompactStringLiteralFormat 0))
       );;
  >>

%%
```

Parser and PrettyPrinter Definition for Algebra domain

What follows is the exact definition of the grammar provided to DMS for the Algebra domain. Fundamentally it is a classic context-free grammar. Note that DMS doesn't use an extended BNF (such as Kleene * or +); those don't add a lot in expressability and it turns out they complicate other computations over ASTs which DMS provides. We've been living without them for 15 years and it just isn't a loss. The grammar terms used are either lexemes as defined (and exactly as spelled) in the lexical definition, or nonterminals appropriate for the domain.

No special machinery is needed to build ASTs. DMS automatically determines how to build ASTs from this grammar, and constructs as it parses. There is no YACC cruft here, no way to get the tree shape wrong, and if you decide to change the grammar, no conceptual or implementation cost except for changing the grammar rules themselves. (The latter truly matters when you have a large language/domain definition.) Grammar changes are truly easy to do.

Interleaved in the grammar definition are *prettyprinting* rules. These tell DMS how to regenerate source text from the internal ASTs that the DMS's parser produces, or more importantly, from ASTs that are generated by transforming other ASTs using the transformation rules. (The final result shown on this page comes from DMS's prettyprinter). We used the basic rules **H*** and **V*** to tell the prettyprinter to print the generated elements of a corresponding AST for a rule, either **H**orizontally, or **V**ertically aligned at the last indentation level. Because we are defining simple mathmatics, we have no need for more sophisticated indentation policies, but DMS has a quite sophisticated language for expressing aligned "box layouts" for prettyprinted source text.

```
-- Algebra.atg
-- Copyright (C) 2010 Semantic Designs; All Rights Reserved

equations = ;
equations = equations equation ;
<<PrettyPrinter>>: { V*; }

equation = sum ;

sum = product ;
[property=associative-commutative] sum = sum '+' product ;
<<PrettyPrinter>>: { H*; }
sum = sum '-' product ;
<<PrettyPrinter>>: { H*; }

product = term ;
[property=associative-commutative] product = product '*' term ;
<<PrettyPrinter>>: { H*; }
product = product '/' term ;
<<PrettyPrinter>>: { H*; }

term = primary ;
term = primary '^' term ;
<<PrettyPrinter>>: { H*; }
```

```
primary = NUMBER ;
primary = NAME ;
primary = '(' sum ')' ;
<<PrettyPrinter>>: { H*; }
primary = '-' primary ;
<<PrettyPrinter>>: { H*; }
primary = NAME '(' sum ')' ; -- allows sin(x), ...
<<PrettyPrinter>>: { H*; }

primary = 'S' NAME '=' sum ':' primary ; -- substititute
<<PrettyPrinter>>: { H*; }

primary = 'D' NAME ':' primary ; -- differentiate primary
<<PrettyPrinter>>: { H*; }
primary = 'I' sum 'D' NAME ; -- indefinite integral
<<PrettyPrinter>>: { H*; }
primary = 'I' sum ',' sum ':' sum 'D' NAME ; -- definite integral
<<PrettyPrinter>>: { H*; }
```

Transformation Rules for Algebra

What follows below this discussion is the exact input used to drive DMS to transform Algebra examples. What is being defined here are *source-to-source* transformation rules. These say, in essence, *if you see X, replace it by Y*. This implements the foundational idea from mathematics of equality: if A is equal to B, one can substitute B for A.

The actual rule format is:

rule *rule_name*(*metavariable_constraints*):*nonterminal*->*nonterminal*
= "*match_pattern*" -> "*replacement_pattern*" **if** *condition*;

Individual rules are named so we can talk about them (e.g., use them, group them, etc.) since in real applications there can be quite a lot of them. The *patterns* are written using the surface syntax of the defined language (Algebra) augumented with metavariables. Surface syntax patterns in rules are written inside meta-domain quotes "...". Meta-escapes are the \ character, and metavariables representing subtrees are then written as *identifier*; each metavariable is constrained to match the syntax category specified for that *identifier* in the rule head. The *condition* places extra, often semantic, constraints on whether a rule can apply, above and beyond pure syntax matching.

The rule **times_self** below is instructive. It only applied to ASTs which represent *product* nonterminals, and it produces a *product* nonterminal (when translating from one language to another, the source and target nonterminals may be very different). The pattern looks for two subtrees which are connected by a multiplication (tree); further, because we used the same metavariable in both places, both subtrees must be identical. The replacement pattern builds a tree involving the exponentiation operator. The condition in this rule prevents it from firing if subtree \x happens to be just a NUMBER; we added this condition to prevent the expression 2*2 from being converted into 2^2.

```
rule times_self(x:term):product -> product
 = " \x * \x " -> " \x ^ 2 " if ~ [z:NUMBER. "\x"<:"\z"];
```

You can see metaprogram control in the forms of **rulesets**. These group rules together, enabling a ruleset to be used as a monolith. Rulesets can be combined to make bigger rules; see the Simplication ruleset.

DMS reads these rules using the parser generated for Algebra, and stores them away for later use by a DMS application.

It is fundamentally important to understand that these rules are not text-string matching and substitution. Rather, they specify, the external string form of the corresponding ASTs that they represent. The transformation process matches the *tree patterns* that the rules specify, against the parsed tree of an instance text processed by DMS; when a match occurs, the matched tree is replaced by the tree corresponding to the right hand side of the rule. Consequence: the pattern matching is perfect; it can only match the proper AST nodes of interest.

These rules should be very unsurprising; they are what you learned in 9th grade algebra, written down in precise form. What's nice is that you write them down so nicely!

```
-+ SimplificationRules.rsl -- Contains rewrite rules to simplify algebra equations
-+ Copyright (C) 2010 Semantic Designs; All Rights Reserved

default base domain Algebra;

rule add_zero(x:sum):sum -> sum
  = " \x + 0 " -> " \x ";

rule subtract_zero(x:sum):sum -> sum
  = " \x - 0 " -> " \x ";

rule subtract_self(x:product): sum->sum
  = " \x - \x " -> " 0 ";

rule minus_minus(x:primary):primary -> primary
  = " - - \x " -> " \x ";

rule times_zero(x:product):product -> product
  = " \x * 0 " -> " 0 ";

rule times_one(x:product):product -> product
  = " \x * 1 " -> " \x ";

rule times_self(x:term):product -> product
  = " \x * \x " -> " \x ^ 2 " if ~ [z:NUMBER. "\x"<:"\z"];

rule divide_one(x:product):product -> product
  = " \x / 1 " -> " \x ";

rule divide_self(x:term):product -> product
  = " \x / \x " -> " 1 ";
```

```
rule divide_by_reciprocal(x:product,y:product,z:term):product -> product
  = " \x / ( \y / \z)  " -> " \x * \z / \y ";

rule power_zero(x:primary):term -> term
  = " \x ^ 0 " -> " 1 ";

rule power_one(x:primary):term -> term
  = " \x ^ 1 " -> " \x ";

rule power_divide_self(x:primary,p:term):product->product
  = " \x ^ \p / \x " -> " \x ^ ( \p - 1 ) ";

rule power_divide_power(x:primary,p:term,q:term):product->product
  = " \x ^ \p / \x ^ \q " -> " \x ^ ( \p - \q )";

rule power_negative(x:primary,y:primary):term -> term
  = " \x ^ - \y " -> " ( 1 / \x ^ \y )";

public ruleset simplify =
   arithmetic +
   remove_useless_parentheses+
   { add_zero,
     subtract_zero,
     subtract_self,
     minus_minus,
     times_zero,
     times_one,
     times_self,
     divide_one,
     divide_self,
     divide_by_reciprocal,
     power_zero,
     power_one,
     power_divide_self,
     power_divide_power,
     power_negative,
     multiplicative_distributive_law2,
     multiplicative_distributive_law3
   };

rule remove_parentheses_sum(x:sum):sum -> sum
  = " ( \x ) " -> " \x ";
rule remove_parentheses_product(x:product):product -> product
  = " ( \x ) " -> " \x ";
rule remove_parentheses_term(x:term):term -> term
  = " ( \x ) " -> " \x ";
rule remove_parentheses_primary(x:primary):primary -> primary
  = " ( \x ) " -> " \x ";

rule add_parentheses_plus1(x:sum,y:product): sum -> sum
  = " \x + ( \y ) " -> " \x + \y ";
rule add_parentheses_plus2(x:sum,y:sum,z:product): sum -> sum
  = " \x + ( \y + \z ) " -> " \x + ( \y ) + \z ";
rule add_parentheses_plus3(x:sum,y:sum,z:product): sum -> sum
  = " \x + ( \y - \z ) " -> " \x + ( \y ) - \z ";
```

```
rule add_parentheses_minus1(x:sum,y:product): sum -> sum
  = " \x - ( \y ) " -> " \x - \y ";
rule add_parentheses_minus2(x:sum,y:sum,z:product): sum -> sum
  = " \x - ( \y + \z ) " -> " \x - \z - ( \y )";
rule add_parentheses_minus3(x:sum,y:sum,z:product): sum -> sum
  = " \x - ( \y - \z ) " -> " \x + \z - ( \y )";

rule multiply_parentheses_multiply1(x:product,y:term): product -> product
  = " \x * ( \y ) " -> " \x * \y ";
rule multiply_parentheses_multiply2(x:product,y:product,z:term): product -> product
  = " \x * ( \y * \z ) " -> " \x * ( \y ) * \z ";
rule multiply_parentheses_multiply3(x:product,y:product,z:term): product -> product
  = " \x * ( \y / \z ) " -> " \x * ( \y ) / \z ";

rule multiply_parentheses_divide1(x:product,y:term): product -> product
  = " \x / ( \y ) " -> " \x / \y ";
rule multiply_parentheses_divide2(x:product,y:product,z:term): product -> product
  = " \x / ( \y * \z ) " -> " \x / \z / ( \y ) ";
rule multiply_parentheses_divide3(x:product,y:product,z:term): product -> product
  = " \x / ( \y / \z ) " -> " \x * \z /( \y ) ";

rule multiplicative_distributive_law2(x:product,y:sum,z:product): sum->sum
  = " \x * ( \y + \z ) " -> " \x * ( \y ) + \x * ( \z  )" ;
rule multiplicative_distributive_law3(x:product,y:sum,z:product): sum->sum
  = " \x * ( \y - \z ) " -> " \x * ( \y ) - \x * ( \z ) " ;

public ruleset remove_useless_parentheses =
   { remove_parentheses_sum,
     remove_parentheses_product,
     remove_parentheses_term,
     remove_parentheses_primary,
     add_parentheses_plus1,
     add_parentheses_plus2,
     add_parentheses_plus3,
     add_parentheses_minus1,
     add_parentheses_minus2,
     add_parentheses_minus3,
     multiply_parentheses_multiply1,
     multiply_parentheses_multiply2,
     multiply_parentheses_multiply3,
     multiply_parentheses_divide1,
     multiply_parentheses_divide2,
     multiply_parentheses_divide3
   };

external pattern Add(x:NUMBER,y:NUMBER):NUMBER
  = 'Support/Naturals/Add' in domain DMSRuleSpecificationLanguage;

external pattern Subtract(x:NUMBER,y:NUMBER):NUMBER
  = 'Support/Naturals/Subtract' in domain DMSRuleSpecificationLanguage;

external pattern Multiply(x:NUMBER,y:NUMBER):NUMBER
  = 'Support/Naturals/Multiply' in domain DMSRuleSpecificationLanguage;

rule fold_constants_add2(x:NUMBER,y:NUMBER):sum -> sum
  = " \x + \y " -> " \Add\(\x\,\y\)";
```

```
rule fold_constants_add3(s:sum,x:NUMBER,y:NUMBER):sum -> sum
  = " \s + \x + \y " -> " \s + \Add\(\x\,\y\)";

rule fold_constants_subtract2(x:NUMBER,y:NUMBER):sum -> sum
  = " \x - \y " -> " \Subtract\(\x\,\y\)";

rule fold_constants_subtract3(s:sum,x:NUMBER,y:NUMBER):sum -> sum
  = " \s + \x - \y " -> " \s + \Subtract\(\x\,\y\)";

rule fold_constants_multiply2(x:NUMBER,y:NUMBER):product -> product
  = " \x * \y " -> " \Multiply\(\x\,\y\)";

rule fold_constants_multiply3(p:product,x:NUMBER,y:NUMBER):product -> product
  = " \p * \x * \y " -> " \p * \Multiply\(\x\,\y\)";

public ruleset arithmetic =
   { fold_constants_add2,
     fold_constants_add3,
     fold_constants_subtract2,
     fold_constants_subtract3,
     fold_constants_multiply2,
     fold_constants_multiply3
   };
```

The above rules implement basic algebraic simplification. We don't claim completenees; this is a demo small enough for an interested reader to absorb.

The following ruleset implement differentiation over algebraic equations. Similarly, we don't make any claim of completeness here. We hope it is pretty obvious how differentiation is implemented.

```
-+ CalculusRules.rsl -- contain rewrites related to simple calculus

default base domain Algebra;

rule derivative_of_constant(x:NAME,c:NUMBER):primary->primary
  = " D \x : \c " -> " 0 ";

rule derivative_of_self(x:NAME):primary->primary
  = " D \x : \x " -> " 1 ";

rule derivative_of_not_self(x:NAME,y:NAME):primary->primary
  = " D \x : \y " -> " 0 " if y ~= x ;

rule derivative_of_power(x:NAME,p:term):primary->primary
  = " D \x : ( \x ^ \p ) " -> "  ( \p * x ^ ( \p -1 ) ) ";

rule derivative_of_sums(x:NAME,s:sum,p:product):primary->primary
  = " D \x : ( \s + \p ) " -> " ( D \x : ( \s ) + D \x : ( \p ) ) ";

rule derivative_of_constant_times(x:NAME,c:NUMBER,t:term):primary->primary
  = " D \x : ( \c * \t ) " -> " ( \c * D \x : ( \t ) ) ";

external ruleset simplify = 'Tools/RuleApplier/SimplificationRules/simplify';
```

```
public ruleset differentiate =
  simplify+
  { derivative_of_constant,
    derivative_of_self,
    derivative_of_not_self,
    derivative_of_constant_times,
    derivative_of_power,
    derivative_of_sums
  };
```

Using DMS to manipulate algebraic equations

The above information is all definitional; it is provided to DMS to give it the ability to parse, simplify, differentiate, and print out results. In this section, we run DMS with these definitions. For our example, we build a formal document (a single file) containing the following "Algebra" domain equations specifying that we want to differentiate a series of formulas:

```
Dx:1

Dx:x

Dx:y

Dx:(8*x^2)+2*y

Dx:(x^y+y)

Dx:(x^e+2*x^2)
```

DMS, using the provided domain definitions, parses these equations into ASTs. If we then applying the Calculus ruleset followed by the Simplification ruleset, and prettyprint the result, DMS produces a series of results (actually, a single tree consisting of the nonterminal *equations*):

```
0

1

0

x * 16    + 2 * y

      y * x ^ ( y - 1 )

      e * x ^ ( e - 1 ) + x * 4
```

The first three examples are intended to be obviously trivial. The fourth example shows that the derivative operator works on a simple polynomial, and is only applied to the *primary* and not to the entire *sum*; this is just a consequence of how the grammar is defined. The fourth example also shows constant folding. The sixth example shows differentiation of a *sum*, and the use of several of the rules to both differentiate, simplify, and apply constant arithmetic.

The odd indentation is a consequence of DMS trying to preserve token positions in the original source code. The column position of the **2** in the third equation, and the **e** in the last equation have been preserved because they were not changed by the transformations, and other tokens were printed after those. In much larger systems of text, this column preservation is useful, to prevent disturbing text nearby the point of a transformation. Should one decide that such column preservation is not right, one can ask DMS to prettyprint the formulas without regard to their original positions. We didn't do that in this case, since the example is produced without anything other than the information on this page.

DMS and formal systems, big and small

This example is intended to be pedagogical and yet show some of the power of DMS. The demonstration uses "algebra" rather than a real computer programming language but from DMS's point of view they are just both formal systems of syntax and so all the ideas apply to tools for specifications and programming languages. (Yes, its rather easy to build simplifiers over expressions in real computer programs, too!)

We have not shown lots of interesting DMS capabilities: predefined language definitions for complex, extant languages, handling of huge systems of files/trees (our example here is 10 lines; DMS is often used on 10 million line systems), construction of symbol tables, control and data flow analysis, conditioning of transformations on these analyses, etc.

How much of this power is needed for a particular program analysis/generation /transformation task depends on the specific task and how much you understand about the problem. The real value in DMS is that it provides a huge amount of machinery, so that when you ultimately discover some machinery is needed for your task, that machinery is available and ready. If you choose a much simpler foundation to solve a problem, and you hit a wall, the cost of switching to another solution (rebuilding parsers, analyzers, ...) is extremely high. Our goal with DMS is to ensure that you never hit such a wall.

- Request a Free Migration Consultation

Search SD

Topics

- Re-engineering
- Documentation
- Assessment
- Improvement
- Code Generation
- Hardware Description Languages
- All Topics

Language:
ECMA/Java script
Product:
Formatter Go!

Semantic Designs- Our Goal

To enable our customers to produce and maintain timely, robust and economical software by providing world-class Software Engineering tools using deep language and problem knowledge with high degrees of automation.
For more information: info@semanticdesigns.com Follow us at Twitter: @SemanticDesigns

Copyright 1995-2023 Semantic Designs, Incorporated
DMS, "Design Maintenance System" and Refactor++ are registered trademarks of Semantic Designs, Inc.
The SD logo and "Semantic Designs" are registered service marks of Semantic Designs, Inc.
Software Reengineering Toolkit, CloneDR, PARLANSE, JOVIAL2C, Thicket, Smart Differencer, CheckPointer are trademarks of Semantic Designs, Inc.
The OMG logo is a registered trademark of the Object Management Group, Inc. in the United States and other countries.
To view our Privacy Policy, click here

Comments or problems: webmaster@semanticdesigns.com
Algebra
as a DMS Domain